

Franz Baader (Ed.)

LNCS 4533

Term Rewriting and Applications

18th International Conference, RTA 2007
Paris, France, June 2007
Proceedings

 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Moshe Y. Vardi

Rice University, Houston, TX, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Franz Baader (Ed.)

Term Rewriting and Applications

18th International Conference, RTA 2007
Paris, France, June 26-28, 2007
Proceedings

Volume Editor

Franz Baader
TU Dresden
Theoretical Computer Science
01062 Dresden, Germany
E-mail: baader@tcs.inf.tu-dresden.de

Library of Congress Control Number: 2007929743

CR Subject Classification (1998): F.4, F.3.2, D.3, I.2.2-3, I.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-73447-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-73447-5 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12086580 06/3180 5 4 3 2 1 0

Preface

This volume contains the papers presented at the 18th International Conference on Rewriting Techniques and Applications (RTA 2007), which was held during June 26–28, 2007, on the campus of the Conservatoire National des Arts et Métiers (CNAM) in Paris, France.

RTA is the major forum for the presentation of research on all aspects of rewriting. Previous RTA conferences were held in Dijon (1985), Bordeaux (1987), Chapel Hill (1989), Como (1991), Montreal (1993), Kaiserslautern (1995), Rutgers (1996), Sitges (1997), Tsukuba (1998), Trento (1999), Norwich (2000), Utrecht (2001), Copenhagen (2002), Valencia (2003), Aachen (2004), Nara (2005), and Seattle (2006).

For RTA 2007, 24 regular papers and 3 system descriptions were accepted for publication out of 69 submissions. Each submission was reviewed by at least three expert reviewers, and an electronic Program Committee (PC) meeting was held on the Internet, using Andrei Voronkov’s *EasyChair* system. The reviews were written by the 14 PC members and 131 additional reviewers, who are listed in these proceedings. I would like to thank the PC members and the additional reviewers for doing such a great job in writing high-quality reviews in time and participating in the electronic PC discussion.

The RTA programme also included three invited talks, by Xavier Leroy (Formal Verification of an Optimizing Compiler), Robert Nieuwenhuis (Challenges in Satisfiability Modulo Theories), and Frank Pfenning (On a Logical Foundation for Explicit Substitutions). The talk by Frank Pfenning was a joint invited talk of RTA and the collocated Eighth International Conference on Typed Lambda Calculi and Applications (TLCA 2007).

The RTA PC decided to award a prize of 1,000 euro for the best paper to the article “On Linear Combinations of λ -Terms” by Lionel Vaux. Moreover, several travel grants could be given to students.

RTA 2007 was held as part of the Federated Conference on Rewriting, Deduction, and Programming (RDP), together with the following events:

- The Eighth International Conference on Typed Lambda Calculi and Applications (TLCA 2007)
- The colloquium From Type Theory to Morphologic Complexity in honor of Giuseppe Longo
- The workshop on Higher Order Rewriting (HOR)
- The workshop on Proof Assistants and Types in Education (PATE)
- The workshop on Rule-Based Programming (RULE)
- The workshop on Security and Rewriting Techniques (SecReT)
- The workshop on Unification (UNIF)

- The workshop on Functional and (Constraint) Logic Programming (WFLP)
- The workshop on Reduction Strategies in Rewriting and Programming (WRS)
- The workshop on Termination (WST)

Many people helped to make RTA 2007 a success. In particular, I would like to thank the Conference Chairs, Ralf Treinen and Xavier Urbain, as well as the rest of the local organization team, and the sponsors of RDP 2007:

- The Conservatoire des Arts et Métiers (CNAM)
- The Centre National de la Recherche Scientifique (CNRS)
- The École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIEE)
- The GDR Informatique Mathématique
- The Institut National de Recherche en Informatique et Automatique (INRIA) unit Futurs
- The Région Île de France

Barbara Morawska and the *EasyChair* system helped to produce the camera-ready copy of these proceedings.

April 2007

Franz Baader

Vincent van Oostrom
Ashish Tiwari
Maribel Fernández
Bernhard Gramlich

Utrecht
Menlo Park
London
Vienna

External Reviewers

Andreas Abel
Beatriz Alarcón
María Alpuente
Roberto M. Amadio
Oana Andrei
Sergio Antoy
Takahito Aoto
Zena Ariola
Pablo Arrighi
Emilie Balland
Alexandru Berlea
Frederic Blanqui
Stefan Blom
Eduardo Bonelli
Iovka Boneva
Guillaume Bonfante
H.J. Sander Bruggink
Roberto Bruni
Antonio Bucciarelli
Wilfried Buchholz
Guillaume Burel
Sergiu Bursuc
Anne-Cécile Caron
Yannick Chevalier
Adam Chlipala
Manuel Clavel
Dario Colazzo
Evelyne Contejean
Andrea Corradini
Rene David
Philippe Devienne
Kevin Donnelly
Gilles Dowek
Francisco Durán
Rachid Echahed
Santiago Escobar
Jerôme Euzenat
Stephan Falke

Emmanuel Filiot
Thomas Genet
Alfons Geser
Jean Goubault-Larrecq
Bernhard Gramlich
Yves Guiraud
Raúl Gutiérrez
Peter Habermehl
Florian Haftmann
Michael Hanus
Tobias Heindel
Joe Hendrix
Miki Hermann
Thomas Hildebrandt
Clement Houtmann
Samuel Hym
Florent Jacquemard
Deepak Kapur
Benny George Kenkireth
Delia Kesner
Jeroen Ketema
Zurab Khasidashvili
Konstantin Korovin
Sava Krstic
Yves Lafont
Francois Lamarche
Julia Lawall
Aurélien Lemay
Joachim Niehren
Stéphane Lengrand
Christof Loeding
Denis Lugiez
Ian Mackie
Yitzhak Mandelbaum
Jacopo Mantovani
Claude Marché
Ralph Matthes
Francois Metayer

Sebastian Mödersheim
Georg Moser
Leonardo de Moura
Paliath Narendran
Enrica Nicolini
Joachim Niehren
Karl-Heinz Niggl
Thomas Noll
Albert Oliveras
Mizuhito Ogawa
Hitoshi Ohsaki
Vincent van Oostrom
Peter Padawitz
Vincent Padovani
Miguel Palomino
Ricardo Peña
Detlef Plump
François Pottier
Pierre Rety
Frank Raiser
Silvio Ranise
Didier Remy
Christian Retoré
Adrian Riesco
Mario Rodriguez-Artalejo
Yves Roos
Grigore Roşu
David Sabel

Masahiko Sakai
Manfred Schmidt-Schauß
Aleksy Schubert
Traian Serbanuta
Jakob Grue Simonsen
Isabelle Simplot-Ryl
Sergei Soloviev
Mark-Oliver Stehr
Toshinori Takai
Jean-Marc Talbot
Alwen Tiu
Marc Tommasi
Xavier Urbain
Christian Urban
Rafael del Vado Virseda
Alberto Verdejo
René Vestergaard
Germán Vidal
Alicia Villanueva
Eelco Visser
Fer-Jan de Vries
Roel de Vrijer
Johannes Waldmann
Edwin Westbrook
Hongwei Xi
Hans Zantema
Francesco Zappa Nardelli

Table of Contents

Formal Verification of an Optimizing Compiler	1
<i>Xavier Leroy</i>	
Challenges in Satisfiability Modulo Theories	2
<i>Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio</i>	
On a Logical Foundation for Explicit Substitutions	19
<i>Frank Pfenning</i>	
Intruders with Caps	20
<i>Siva Anantharaman, Paliath Narendran, and Michael Rusinowitch</i>	
Tom: Piggybacking Rewriting on Java	36
<i>Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles</i>	
Rewriting Approximations for Fast Prototyping of Static Analyzers	48
<i>Yohan Boichut, Thomas Genet, Thomas Jensen, and Luka Le Roux</i>	
Determining Unify-Stable Presentations	63
<i>Thierry Boy de la Tour and Mnacho Echenim</i>	
Confluence of Pattern-Based Calculi	78
<i>Horatiu Cirstea and Germain Faure</i>	
A Simple Proof That Super-Consistency Implies Cut Elimination	93
<i>Gilles Dowek and Olivier Hermant</i>	
Bottom-Up Rewriting Is Inverse Recognizability Preserving	107
<i>Irène Durand and Géraud Sénizergues</i>	
Adjunction for Garbage Collection with Application to Graph Rewriting	122
<i>Dominique Duval, Rachid Echahed, and Frederic Prost</i>	
Non Strict Confluent Rewrite Systems for Data-Structures with Pointers	137
<i>Rachid Echahed and Nicolas Peltier</i>	
Symbolic Model Checking of Infinite-State Systems Using Narrowing . . .	153
<i>Santiago Escobar and José Meseguer</i>	
Delayed Substitutions	169
<i>José Espírito Santo</i>	

Innermost-Reachability and Innermost-Joinability Are Decidable for Shallow Term Rewrite Systems	184
<i>Guillem Godoy and Eduard Huntingford</i>	
Termination of Rewriting with Right-Flat Rules	200
<i>Guillem Godoy, Eduard Huntingford, and Ashish Tiwari</i>	
Abstract Critical Pairs and Confluence of Arbitrary Binary Relations . . .	214
<i>Rémy Haemmerlé and François Fages</i>	
On the Completeness of Context-Sensitive Order-Sorted Specifications	229
<i>Joe Hendrix and José Meseguer</i>	
KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis	246
<i>Mark Hills and Grigore Roşu</i>	
Simple Proofs of Characterizing Strong Normalization for Explicit Substitution Calculi	257
<i>Kentaro Kikuchi</i>	
Proving Termination of Rewrite Systems Using Bounds	273
<i>Martin Korp and Aart Middeldorp</i>	
Sequence Unification Through Currying	288
<i>Temur Kutsia, Jordi Levy, and Mateu Villaret</i>	
The Termination Competition	303
<i>Claude Marché and Hans Zantema</i>	
Random Descent	314
<i>Vincent van Oostrom</i>	
Correctness of Copy in Calculi with Letrec	329
<i>Manfred Schmidt-Schauß</i>	
A Characterization of Medial as Rewriting Rule	344
<i>Lutz Straßburger</i>	
The Maximum Length of Mu-Reduction in Lambda Mu-Calculus	359
<i>Makoto Tatsuta</i>	
On Linear Combinations of λ -Terms	374
<i>Lionel Vaux</i>	
Satisfying KBO Constraints	389
<i>Harald Zankl and Aart Middeldorp</i>	
Termination by Quasi-periodic Interpretations	404
<i>Hans Zantema and Johannes Waldmann</i>	
Author Index	419

Formal Verification of an Optimizing Compiler

Xavier Leroy

INRIA Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
`Xavier.Leroy@inria.fr`

Programmers naturally expect that compilers and other code generation tools produce executable code that behaves as prescribed by source programs. However, compilers are complex programs that perform many subtle transformations. Bugs in compilers do happen and can lead to silently producing incorrect executable code from a correct source program. This is a significant concern in the context of high-assurance software that has been verified (at the source level) using formal methods (static analysis, model checking, program proof, etc): any bug in the compiler can potentially invalidate the guarantees so painfully established by the use of formal methods.

There are several ways to generate confidence in the compilation process, including translation validation and proof-carrying code. This talk focuses on applying program proof technology to the compiler itself, in order to prove a semantic preservation theorem for every pass of the compiler. We present preliminary results from the Compcert experiment: the development and proof of correctness of a moderately-optimizing compiler for a large subset of the C language. The proof of correctness is mechanized using the Coq proof assistant. Moreover, most of the compiler itself is written directly in the functional subset of the Coq specification language, from which executable Caml code is automatically extracted.

The preliminary results are encouraging and suggest two directions for long-term research. One is the formal verification of other tools (code generators, static analyzers, provers, ...) involved in the production and certification of high-assurance software. The other is the systematic use of proof assistants to mechanize programming language semantics, type systems, program transformations and related formal systems.

References

1. Bertot, Y., Grégoire, B., Leroy, X.: A structured approach to proving compiler optimizations based on dataflow analysis. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 66–81. Springer, Heidelberg (2006)
2. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 460–475. Springer, Heidelberg (2006)
3. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, pp. 42–54. ACM Press, New York (2006)

Challenges in Satisfiability Modulo Theories

Robert Nieuwenhuis, Albert Oliveras,
Enric Rodríguez-Carbonell, and Albert Rubio*

Abstract. Here we give a short overview of the DPLL(T) approach to Satisfiability Modulo Theories (SMT), which is at the basis of current state-of-the-art SMT systems. After that, we provide a documented list of theoretical and practical current challenges related to SMT, including some new ideas to exploit SAT techniques in Constraint Programming.

1 Introduction

Propositional satisfiability checkers (SAT solvers) are currently being applied in more and more contexts, including hardware and software verification, in Operations Research (planning, scheduling), as well as in Biology, Linguistics and Medicine. Most SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62]. The performance of DPLL-based SAT solvers has improved spectacularly in the last years, due to better implementation techniques and conceptual enhancements such as *backjumping*, *conflict-driven lemma learning* and *restarts* [MSS99, MMZ⁺01, ES03]. However, some practical problems are more naturally expressed in logics that are more expressive than propositional logic.

For example, for timed automata, a good choice is *difference logic*, where formulas contain atoms of the form $a - b \leq k$, which are interpreted with respect to a background theory T of the integers, rationals or reals. Similarly, for the verification of pipelined microprocessors it is convenient to consider a logic of *Equality with Uninterpreted Functions (EUF)*, where the background theory T specifies a congruence [BD94]. To mention just one other example, the conditions arising from program verification usually involve arrays, lists and other data structures, so it becomes very natural to consider satisfiability problems *modulo* the theory T of these data structures. In such applications, problems may contain thousands of clauses like

$$p \vee \neg q \vee a = f(b - c) \vee \text{read}(s, f(b - c)) = d \vee a - g(c) \leq 7$$

containing purely propositional atoms as well as atoms over (combined) theories. This is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T : given a formula F , determine whether F is T -satisfiable, i.e., whether there exists a model of T that is also a model of F .

* Technical Univ. of Catalonia, Barcelona. All authors partially supported by Spanish Min. of Educ. and Science through the LogicTools project (TIN2004-03382) and Intel Corp. Research Grant: “SMT Solvers for High-Level Hardware Verification”.

SMT has become an extremely active area of research. A rapidly growing library of benchmarks for SMT with a formal syntax and semantics exists [RT03], as well as a yearly SMT competition (both SMT-LIB and SMT-COMP are easily found on the web).

The DPLL(T) approach to SMT is based on a general DPLL(X) engine, whose parameter X can be instantiated with specialized solvers $Solver_T$ for given theories T , thus producing a system DPLL(T). Once the DPLL(X) engine has been implemented, new theories can be dealt with by simply plugging in new theory solvers. These solvers must only be able to deal with *conjunctions* of theory literals and conform to a minimal and simple set of additional requirements.

In Sections 2, 3 and 4 of this paper, by means of a rewrite-rule-based framework called Abstract DPLL we first give a brief overview of DPLL, SMT, and the DPLL(T) approach to SMT (we refer to [NOT06] for all details and references). In Section 5 we describe a number of theoretical and practical challenges in SMT. Extensions for handling new theories and applications, including optimization and constraint programming are discussed, as well as for first-order theorem proving. Other challenges involve the design of efficient data structures and algorithms for implementing certain key parts of SMT solvers. All of them are closely related to the area of rewriting.

2 The DPLL Procedure

Let P be a fixed finite set of propositional symbols. If $p \in P$, then p and $\neg p$ are *literals* of P . The *negation* of a literal l , written $\neg l$, denotes $\neg p$ if l is p , and p if l is $\neg p$. A *clause* is a disjunction of literals $l_1 \vee \dots \vee l_n$. A *unit clause* is a clause consisting of a single literal. A (finite, non-empty, CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \dots \wedge C_n$. When it leads to no ambiguities, we sometimes also write such a formula in set notation $\{C_1, \dots, C_n\}$ or simply replace \wedge connectives by commas.

A (partial truth) *assignment* M is a set of literals such that $\{p, \neg p\} \subseteq M$ for no p . A literal l is *true* in M if $l \in M$, it is *false* in M if $\neg l \in M$, and l is *undefined* in M otherwise. M is *total* over P if no literal of P is undefined in M . A clause C is true in M if at least one of its literals is true in M . It is false in M if all its literals are false in M , and it is undefined in M otherwise. A formula F is true in M , or *satisfied* by M , denoted $M \models F$, if all its clauses are true in M . In that case, M is called a *model* of F . If F has no models then it is called *unsatisfiable*. If F and F' are formulas, we write $F \models F'$ if F' is true in all models of F . Then we say that F' is *entailed* by F , or is a *logical consequence* of F .

In what follows, (possibly subscripted or primed) lowercase l *always* denotes literals. Similarly C and D always denote clauses, F and G denote formulas, and M and N are assignments. If C is a clause $l_1 \vee \dots \vee l_n$, we sometimes write $\neg C$ to denote the formula $\neg l_1 \wedge \dots \wedge \neg l_n$.

Here a DPLL procedure is modeled by a transition relation over states. A state is either *FailState* or a pair $M \parallel F$, where F is a finite set of clauses and M is a sequence of literals that is seen as a partial assignment. Some literals l

in M will be *annotated* as being *decision literals*; these are the ones added to M by the Decide rule given below, and are sometimes written l^d . The transition relation is defined by means of rules.

Definition 1. *The DPLL system with Backtrack consists of the four rules:*

UnitPropagate :

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

Decide :

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

Fail :

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

Backtrack :

$$M l^d N \parallel F, C \implies M \neg l \parallel F, C \quad \text{if} \quad \begin{cases} M l^d N \models \neg C \\ N \text{ contains no decision literals} \end{cases}$$

One can use these rules for deciding the satisfiability of an input CNF F by simply generating an arbitrary derivation $\emptyset \parallel F \implies \dots \implies S_n$, where S_n is irreducible by the rules. Such derivations are always finite, and

- (i) F is unsatisfiable if, and only if, the final state S_n is *FailState*, and
- (ii) if S_n is of the form $M \parallel F$ then M is a model of F .

These rules speak for themselves, providing a classical depth-first search with backtracking, where the Decide rule represents a case split: an undefined literal l is chosen from F , and added to M . The literal is annotated as a *decision literal*, to denote that, if $M l^d$ cannot be extended to a model of F , then (by Backtrack) still the other possibility $M \neg l$ must be explored. In the following, if M is a sequence of the form $M_0 l_1 M_1 \dots l_k M_k$, where the l_i are all the decision literals in M , then the literals of each $l_i M_i$ are said to *belong to decision level* i .

Example 2. In the following derivation, to improve readability we have denoted atoms by natural numbers, negation by overlining, and written decision literals in **bold** font:

$$\begin{array}{llllllll}
\emptyset \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(Decide)} \\
\mathbf{1} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(UnitPropagate)} \\
\mathbf{1} \overline{2} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(UnitPropagate)} \\
\mathbf{1} \overline{2} \mathbf{3} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(UnitPropagate)} \\
\mathbf{1} \overline{2} \mathbf{3} \mathbf{4} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(Backtrack)} \\
\overline{1} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(UnitPropagate)} \\
\overline{1} \mathbf{4} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(Decide)} \\
\overline{1} \mathbf{4} \overline{3} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & \implies & \text{(UnitPropagate)} \\
\overline{1} \mathbf{4} \overline{3} \mathbf{2} \parallel & \overline{1}\overline{2}, & 2\vee 3, & \overline{1}\overline{3}\overline{4}, & 2\overline{3}\overline{4}, & 1\vee 4 & & \text{Final state:} \\
& & & & & & & \text{model found.} \quad \square
\end{array}$$

In modern DPLL-based SAT solvers instead of **Backtrack** a more general **Backjump** rule is considered, of which **Backtrack** is a particular case.

Definition 3. *The Backjump rule is defined as follows:*

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

We call the clause $C' \vee \text{l}'$ a *backjump clause*.

Example 4. The aim of this **Backjump** rule is to generalize backtracking by a better analysis of why the so-called *conflicting* clause C is false. Standard backtracking reverses the *last* decision, and adds it (as a non-decision literal) to the previous decision level. Backjumping generalizes this by adding a new literal to a possibly lower decision level. Consider:

$$\begin{array}{llllll} \emptyset & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(Decide)} \\ \mathbf{1} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(UnitPropagate)} \\ \mathbf{1\ 2} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(Decide)} \\ \mathbf{1\ 2\ 3} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(UnitPropagate)} \\ \mathbf{1\ 2\ 3\ 4} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(Decide)} \\ \mathbf{1\ 2\ 3\ 4\ 5} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(UnitPropagate)} \\ \mathbf{1\ 2\ 3\ 4\ 5\ 6} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \text{(Backjump)} \\ \mathbf{1\ 2\ 5} & \parallel & \bar{1}\vee 2, & \bar{3}\vee 4, & \bar{5}\vee \bar{6}, & 6\vee \bar{5}\vee \bar{2} & \implies & \dots \end{array}$$

Before the **Backjump** step, the clause $6\vee \bar{5}\vee \bar{2}$ is conflicting: it is false in $\mathbf{1\ 2\ 3\ 4\ 5\ 6}$. The reason for its falsity is the literal 2 together with the decision 5 and its unit propagation $\bar{6}$. Therefore, one can infer that 2 is incompatible with the decision 5. Indeed, the backjump clause $\bar{2} \vee \bar{5}$ is a logical consequence of the last two clauses. It allows us to return to the first decision level, adding there, as a unit propagation, the literal $\bar{5}$ (which plays the role of l' in the **Backjump** rule). \square

Note that in the previous example an application of **Backtrack** instead of **Backjump** would have given a state with first component $\mathbf{1\ 2\ 3\ 4\ 5}$, even though the decision level $\mathbf{3\ 4}$ is unrelated with the reasons for the falsity of $6 \vee \bar{5} \vee \bar{2}$. Moreover, intuitively, the search state $\mathbf{1\ 2\ 5}$ reached after **Backjump** is more *advanced* than $\mathbf{1\ 2\ 3\ 4\ 5}$. This notion of “being more advanced” is formalized in Theorem 8 below.

The following example shows how **Backjump** can be applied in practice, by finding an adequate backjump clause.

Example 5. Consider a state $M \parallel F$, where, among other clauses, F contains:

$$\bar{9}\vee \bar{6}\vee 7\vee \bar{8} \quad 8\vee 7\vee \bar{5} \quad \bar{6}\vee 8\vee 4 \quad \bar{4}\vee \bar{1} \quad \bar{4}\vee 5\vee 2 \quad 5\vee 7\vee \bar{3} \quad 1\vee \bar{2}\vee 3$$

and M is of the form: $\dots 6 \dots \bar{7} \dots \mathbf{9\ 8\ 5\ 4\ 1\ 2\ 3}$. It is easy to observe how by six applications of **UnitPropagate** this state has been reached after the last

decision **9**. For example, $\bar{8}$ is implied by 9, 6, and $\bar{7}$, due to the leftmost clause $\bar{9} \vee \bar{6} \vee 7 \vee \bar{8}$. The ordered sequence of propagated literals is stored, each one of them together with the clause that caused it. In this state $M \parallel F$, the clause $1 \vee \bar{2} \vee 3$ is conflicting, since M contains $\bar{1}$, 2 and $\bar{3}$. Now one can trace back the reasons for this conflicting clause. For example, $\bar{3}$ was implied by $\bar{5}$ and $\bar{7}$, due to the clause $5 \vee 7 \vee \bar{3}$. The literal $\bar{5}$ was in turn implied by $\bar{8}$ and $\bar{7}$, and so on. In this way, working backwards from the conflicting clause, and in the reverse order in which each literal was propagated, we get the following (*conflict*) *resolution* proof:

$$\begin{array}{c} \underline{\underline{\bar{4} \vee 5 \vee 7 \vee \bar{3} \quad 1 \vee \bar{2} \vee 3}} \\ \bar{4} \vee 5 \vee 2 \quad 5 \vee 7 \vee 1 \vee \bar{2} \\ \underline{\underline{\bar{4} \vee \bar{1} \quad \bar{4} \vee 5 \vee 7 \vee 1}} \\ \bar{6} \vee 8 \vee 4 \quad 5 \vee 7 \vee \bar{4} \\ \underline{\underline{8 \vee 7 \vee \bar{5} \quad \bar{6} \vee 8 \vee 7 \vee 5}} \\ 8 \vee 7 \vee \bar{6} \end{array}$$

The process stops once it generates a clause with only one atom of the current decision level, in our example, the literal 8 in the clause $8 \vee 7 \vee \bar{6}$. This process is equivalent to the *implication-graph-based* conflict analysis, where this literal is called the *First Unique Implication Point (1UIP)*, see [MSS99, MMZ+01]. The clause obtained is a backjump clause $C \vee l'$ where the 1UIP is the literal l' . \square

In most modern DPLL implementations, the backjump clause is always *added to the clause set* as a *learned clause (conflict-driven clause learning)*. In the previous example, learning the lemma $8 \vee 7 \vee \bar{6}$ will allow the application of UnitPropagate to any state where M contains the negation of two of its literals, hence preventing any conflict caused by having the negation of all three. Indeed, reaching such *similar* conflicts frequently happens in industrial problems having some regular structure, and learning such lemmas is known to be very effective. Since a lemma is aimed at preventing future similar conflicts, when such conflicts are not very likely to be found again the lemma can be removed. In practice this is done if its *activity* (e.g., the number of times it is involved in a conflict) has become low.

Definition 6. *The rules of Learn and Forget are the following ones:*

Learn :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \text{if} \quad \begin{cases} \text{all atoms of } C \text{ occur in } F \\ F \models C \end{cases}$$

Forget :

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \text{if} \quad \{F \models C\}$$

State-of-the-art SAT-solvers [MMZ+01, ES03] essentially apply these rules using efficient implementation techniques for UnitPropagate (e.g., watching two literals for unit propagation [MMZ+01]), and *activity-based* heuristics for selecting the decision literal for Decide: split on literals that occur in *recent* lemmas and conflicts. In addition, the DPLL procedure may periodically be *restarted* to

escape from bad search behaviors. The rationale behind this is that upon each **Restart** (i.e., $M \parallel F \implies \emptyset \parallel F$), the newly learned lemmas will lead the heuristics for **Decide** to behave differently, and hopefully cause the procedure to explore the search space in a more compact way. We have the following results for the DPLL rules introduced so far (see [NOT06](#) for all details):

Theorem 7. *If $\emptyset \parallel F \implies^* S$ where S is irreducible w.r.t. **Decide**, **Backjump** and **Fail**, then (i) S is **FailState** if, and only if, F is unsatisfiable, and (ii) if S is of the form $M \parallel F'$ then M is a model of F .*

Theorem 8. *Any derivation $\emptyset \parallel F \implies S_1 \implies \dots$ is finite if it contains only finitely many consecutive **Learn** and **Forget** steps and **Restart** is applied only with increasing periodicity.*

3 Satisfiability Modulo Theories

Now let the set P over which formulas are built be a fixed finite set of *ground* (i.e., variable-free) first-order atoms (instead of propositional symbols as before). A *theory* T is a set of closed first-order formulas that is satisfiable in the first-order sense. A formula F is *T -satisfiable* or *T -consistent* if $F \wedge T$ is satisfiable in the first-order sense. If M is a T -consistent partial assignment and F is a formula such that $M \models F$, i.e., M is a (propositional) model of F , then we say that *M is a T -model of F* . The SMT problem for a theory T is the problem of determining, given a formula F , whether F is T -satisfiable, or, equivalently, whether F has a T -model. Note that, as usual in SMT, here we only consider the SMT problem for *ground* (and hence quantifier-free) CNF formulas F . Also note that F may contain constants that are free in T , which, as far as satisfiability is concerned, can equivalently be seen as existentially quantified variables. We will consider here only theories T such that the T -satisfiability of conjunctions of such ground literals is decidable, and a decision procedure for doing so is called a *T -solver*. If F and G are formulas, then *F entails G in T* , written $F \models_T G$, if $F \wedge \neg G$ is T -inconsistent.

The *eager* approach to SMT is based on sophisticated satisfiability-preserving translations from SMT into SAT. But on many practical problems the translation or the SAT solver run out of time or memory. Alternatively, translating into DNF and using a T -solver for deciding the satisfiability of conjunctions of theory literals is also too inefficient due to the exponential blowup of the DNF.

Therefore, the *lazy* approach tries to combine specialized T -solvers with state-of-the-art SAT solvers for dealing with the boolean structure of the formulas. It initially considers each atom as a propositional symbol, i.e., it “forgets” about the theory T . If a SAT solver reports propositional unsatisfiability, then F is also T -unsatisfiable. If it returns a propositional model of F , then this model (a conjunction of literals) is checked by a T -solver. If it is T -satisfiable then it is a T -model of F . Otherwise, the T -solver builds a ground clause, called a *theory lemma*, a clause C such that $\emptyset \models_T C$, precluding that model. This lemma is added to F and the SAT solver is started again. This process is repeated until

the SAT solver finds a T -satisfiable model or returns unsatisfiable. The lazy approach is quite flexible, by combining any SAT solver with any T -solver. See [\[NOT06\]](#) for a more detailed comparison of approaches and references.

Example 9. Assume we are deciding the satisfiability of a large EUF formula, i.e., the background theory T is equality, and assume that the model M found by the SAT solver contains, among many others, the literals: $b = c$, $f(b) = c$, $a \neq g(b)$, and $g(f(c)) = a$. Then the T -solver detects that M is not a T -model, since $b = c \wedge f(b) = c \wedge g(f(c)) = a \not\models_T a = g(b)$. Therefore, the lazy procedure has to be restarted after the corresponding theory lemma has been added to the clause set. In principle, one can take as theory lemma simply the negation of M , that is, the disjunction of the negations of all the literals in M . However, this clause may therefore have thousands of literals, and the lazy approach will behave much more efficiently if the T -solver is able to generate a small *explanation* of the T -inconsistency of M , which in this example could be the clause $b \neq c \vee f(b) \neq c \vee g(f(c)) \neq a \vee a = g(b)$. \square

3.1 DPLL Modulo Theories

We now adapt the abstract DPLL framework for the propositional case presented in the previous section. Here **Learn**, **Forget** and **Backjump** are slightly modified in order to work modulo theories: in these rules, entailment between formulas now becomes entailment in T :

Definition 10. *The rules T -Learn, T -Forget and T -Backjump are:*

T -Learn :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \text{if} \quad \left\{ \begin{array}{l} \text{every atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{array} \right.$$

T -Forget :

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \text{if} \quad \{ F \models_T C \}$$

T -Backjump :

$$M \text{ l}^d N \parallel F, C \quad \Longrightarrow \quad M \text{ l}' \parallel F, C \quad \text{if} \quad \left\{ \begin{array}{l} M \text{ l}^d N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee \text{ l}' \text{ such that:} \\ F, C \models_T C' \vee \text{ l}' \text{ and } M \models \neg C', \\ \text{l}' \text{ is undefined in } M, \text{ and} \\ \text{l}' \text{ or } \neg \text{l}' \text{ occurs in } F \text{ or in } M \text{ l}^d N \end{array} \right.$$

The naive lazy approach to SMT is modeled as follows using the rules. Each time a state $M \parallel F$ is reached that is irreducible with respect to **Decide**, **Fail** and **T -Backjump**, M can be T -consistent or not. If it is, then M is indeed a T -model of F . If it is not, then there exists a subset $\{l_1, \dots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \dots \vee \neg l_n$. By one T -Learn step, this theory lemma $\neg l_1 \vee \dots \vee \neg l_n$ can

be learned and then **Restart** can be applied. If these theory lemmas are never removed by the T -Forget rule, this strategy is terminating, sound and complete in a similar sense as in the previous section.

Several important enhancements of the lazy approach can now easily be modeled using the rules:

Incremental T -solver. The T -consistency of the model can be checked incrementally, while the model is being built by the DPLL procedure, i.e., without delaying the check until a propositional model has been found, thus saving useless work. Assume a state $M \parallel F$ has been reached such that M is T -inconsistent. Then, as in the naive lazy approach, there exists a subset $\{l_1, \dots, l_n\}$ of M such that $\emptyset \models_T \neg l_1 \vee \dots \vee \neg l_n$. This theory lemma is then learned, reaching the state $M \parallel F, \neg l_1 \vee \dots \vee \neg l_n$. As in the previous case, then **Restart** can be applied.

Incremental T -solver and on-line SAT solver. When a T -inconsistency is detected by the incremental T -solver, the DPLL procedure can simply backtrack to the last point where the assignment was still T -consistent, instead of restarting from scratch. As in the previous case, if a T -inconsistency is detected, a state $M \parallel F, \neg l_1 \vee \dots \vee \neg l_n$ is reached. But now the procedure *repairs* the T -inconsistency of the partial model by exploiting the fact that $\neg l_1 \vee \dots \vee \neg l_n$ is a conflicting clause, and hence either **Fail** or T -Backjump applies.

Theory propagation. In the approach presented so far, the T -solver provides information only *after* a T -inconsistent partial assignment has been generated. In this sense, the T -solver is used only to *validate* the search a posteriori, not to *guide* it a priori. In order to overcome this limitation, the T -solver can also be used in a given DPLL state $M \parallel F$ to detect literals l occurring in F such that $M \models_T l$, allowing the DPLL procedure to move to the state $Ml \parallel F$. This is called *theory propagation*. It requires the following additional rule **Theory Propagate**:

$$M \parallel F \implies Ml \parallel F \text{ if } \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

Exhaustive Theory Propagation. For some theories it even pays off, for every state $M \parallel F$, to eagerly detect and propagate *all* literals l occurring in F such that $M \models_T l$. Then, in every state $M \parallel F$ the model M will be T -consistent, and hence the T -solver will never (need to) detect any T -inconsistencies. It is modeled simply by assuming that **Theory Propagate** is applied eagerly.

Similar correctness, termination, and completeness results apply as given in the previous section for the propositional case (see [NOT06] for details).

4 The DPLL(T) Approach

DPLL(T) is based on a general DPLL engine, called DPLL(X), combined with a module $Solver_T$ that can handle conjunctions of literals in T . This is similar

to the $CLP(X)$ scheme for constraint logic programming: a clean and modular, but efficient, use of specialized solvers within a general-purpose engine. $DPLL(T)$ combines the advantages of the eager and lazy approaches to SMT. As soon as the theory starts playing a significant role, $DPLL(T)$ is very efficient (see the SMT-COMP results), and it has the flexibility of the lazy approaches, by simply plugging in other solvers that conform to a minimal interface.

Here we describe the $DPLL(T)$ approach without exhaustive theory propagation (see [\[NO05a\]](#) for an exhaustive approach for *difference logic*). For the initial setup of $DPLL(T)$, $Solver_T$ reads the input CNF, stores the list of all literals occurring in it, and hands it over to $DPLL(X)$, who treats it as a purely propositional CNF. After that, $DPLL(T)$ implements the rules as follows:

- Each time $DPLL(X)$ communicates to $Solver_T$ that another literal l is added to the partial model M (e.g., due to `UnitPropagate` or to `Decide`), $Solver_T$ answers indicating whether M is still T -consistent. If not, $Solver_T$ returns a (preferably small) *explanation* why, that is, a subset $\{l_1, \dots, l_n\}$ of M that becomes T -inconsistent by adding l to it. $DPLL(X)$ then handles $\neg l_1 \vee \dots \vee \neg l_n$ as a conflicting clause, applying T -Backjump or `Fail`.
- $DPLL(X)$ can also ask $Solver_T$ to return a (possibly incomplete) list of literals that are T -consequences, to which it then applies `Theory Propagate`.
- $DPLL(X)$ applies `Fail` or T -Backjump if after `UnitPropagate` a conflict is detected. For T -Backjump, the backjump clause is built as in [Example 5](#), but with an important difference: a literal l can now be in M not only by `Decide` or by `UnitPropagate`, but also due to an application of `Theory Propagate`. In the last case, the conflict resolution process requires that $Solver_T$ must be able to also give *explanations* of theory propagations, that is, to recover a (preferably small) subset of literals $\{l_1, \dots, l_n\}$ of M that T -entailed l . $DPLL(X)$ then treats l in the resolution process as if $\neg l_1 \vee \dots \vee \neg l_n \vee l$ had caused a unit propagation of l .
- At each T -Backjump application, T -Learn learns the backjump clause (which is a T -consequence of the current formula). $DPLL(X)$ also tells $Solver_T$ how many literals of the partial interpretation have been unassigned in the backjump, which allows $Solver_T$ to undo them.
- $DPLL(X)$ applies `Decide` only if none of `Theory Propagate`, `UnitPropagate`, `Fail` or T -Backjump is applicable. An activity-based heuristic for choosing the decision literal as in propositional DPLL is used.
- In a typical $DPLL(T)$ implementation, $DPLL(X)$ applies `Restart` when certain system parameters reach some prescribed limits, such as the number of conflicts or lemmas, the number of new units derived, etc. T -Forget can be applied, e.g., after each restart, removing part of the lemmas according to their activity (number of times involved in a conflict, etc.). Usually the newest lemmas are not removed.

5 Challenges

We now describe a number of theoretical and practical challenges in SMT.

First we consider extensions for improving the solvers for some of the most important theories: equality, linear arithmetic, and bitvectors. This involves questions of both theoretical and practical nature.

After that, we discuss some challenges arising in the context of the extension of SMT for handling formulas with universal quantifiers, i.e., for first-order theorem proving.

Finally we discuss new ideas for extending SMT to other application areas including optimization and constraint programming. This also involves the development of solvers for new theories.

5.1 Challenges for Improving Current Theory Solvers

Let us first discuss T -solvers for EUF logic, where the theory is just equality (a congruence). As for any T -solver, its requirements are as explained in the previous section: each time an additional literal comes in, it must check whether the conjunction remains T -consistent, and, if not, give an *explanation* (a T -inconsistent subset of the literals); it must also be able to find theory propagations and, when demanded, give explanations of these too; and it must be capable of backtracking, i.e., undoing (dis)equalities.

Positive equality literals can be propagated efficiently by congruence closure (CC) [DST80]. In [NO07] an incremental, backtrackable CC algorithm is given which can also efficiently retrieve explanations from CC, which is non-trivial.

Challenge 1: This challenge was first discussed at this conference in 2005 [NO05]. It is widely understood that small explanations tend to behave better in practice. Finding for CC an explanation with the *minimum number of literals* is NP-hard (Ashish Tiwari, personal communication). Hence minimality w.r.t. \subseteq is considered. The explanations produced in [NO07] may, in a small percentage of cases, contain redundant equations. How to get irredundant ones, or small(er) ones in some other sense? Studying this may produce useful new insights, although it may only have a limited practical impact on the performance of SMT solvers.

Challenge 2: Determine the exact complexity of CC. The aforementioned CC algorithms are $O(n \log n)$, but it is still unknown whether this is optimal. Some researchers conjecture that something like $O(n \alpha(n, n))$, as in Union-Find, might be possible for CC, and hence also for the ground word problem.

Challenge 3: The development of proof-producing SMT solvers is an important research topic. How to do efficient CC proof mining? For more details on this challenge, see [ST05], where Stump and Tan (two years ago at RTA) gave an elegant rewrite-based approach for equivalence closure; see also [SL06], an ingredient for its extension to CC.

Together with EUF logic, so far the most important classes of T -solvers are those for (fragments of) linear arithmetic over the integer or real numbers (see for instance the list of logics in SMT-LIB).

Challenge 4: It is well-known that, for many problems arising from the real world, a non-negligible percentage of the literals in linear arithmetic actually falls into difference logic¹ (see also [BBC⁺05]). This observation cries out for techniques for linear constraint solving that are “difference-logic-aware”. In this direction, promising results have already been accomplished in [DdM06b], thanks to, among others, a simplex procedure with a particular treatment of bound constraints, i.e., of the form $a \leq k$ or $a \geq k$, which in some cases is even faster than specialized tools for difference logic. However, for dense difference-logic problems, such as those coming from scheduling, there is room for improvement [DdM06a].

Challenge 5: There is practical evidence that a good way to handle equalities (respectively, disequalities) is by splitting these constraints as conjunctions (respectively, disjunctions) of inequalities. For instance, in the case of the integers, the satisfiability of a conjunction of difference logic disequalities and inequalities is NP-complete, whereas by restricting to inequalities the problem becomes polynomial; thus, splitting allows one to pass the NP-hardness of the solver to the boolean engine, which is designed to be efficient in handling the search space, as explained in previous sections. In the case of linear real arithmetic, in order to detect inconsistencies with disequalities, it is necessary to detect all implicit equalities implied by the constraints in the assignment, which may entangle a costly overhead; state-of-the-art solvers confirm this fact experimentally [DdM06b]. Therefore, boolean splittings can be exploited to improve efficiency. A natural problem is thus whether there exist new better ways of using the boolean engine in order to simplify the theories and so get faster solvers.

Challenge 6: So far, all SMT tools for full linear arithmetic employ infinite-precision numbers to guarantee the soundness of the results (since most of them are applied in verification applications). Although there exist sophisticated numerical libraries for this purpose, e.g., GMP², the involved overhead must not be neglected. A challenge would be to employ non-precise arithmetic so as to obtain more efficient solvers, as done in the context of Operations Research [ILO]. Is there any clever way of using an efficient non-precise off-the-shelf solver, and then only do a few checks with infinite-precision to guarantee soundness? A possibility could also be to develop solvers based on interior-point algorithms [Ter96, RTV97], which can only be implemented efficiently by means of floating-point arithmetic.

Finally, one of the most challenging theories in SMT, mainly due to its application to hardware verification, is the one of bitvectors. Elements of this domain can be viewed as arrays of bits, to which bitwise logical operators can be applied; but they can also be seen as integers, requiring support for the elementary arithmetic operations.

This inherent duality is also reflected on the existing techniques. On the one hand, translating the problem into propositional logic (known as *bit-blasting*) is

¹ See eecs.berkeley.edu/~sseshia/research/uclid.html.

² See <http://gmplib.org/>.

well-suited for problems where bitwise operators dominate. On the other hand, when the problem has a prevailing arithmetic component, encoding it in linear integer arithmetic is the method of choice.

Challenge 7: Unfortunately, when there is no significant dominance none of the current methods is satisfactory. The challenge is to obtain hybrid procedures that combine the benefits of both approaches, e.g., by bit-blasting only very lazily.

Challenge 8: Are there any fragments of the theory of bitvectors that can be handled more efficiently, but are still useful for certain practical applications?

5.2 Challenges for SMT with Quantifiers

SMT is typically considered to be the problem of checking the satisfiability of a ground first-order formula modulo a background theory T . If a T -solver for this particular theory is available, no quantifier reasoning is necessary at all. However, for several reasons, this is sometimes a too optimistic setting:

- In some applications, the ground fragment is not expressive enough and one needs to introduce first-order quantifiers in the formula. This is the case, just to give an example, in proof obligations arising from software verification where loop invariants may contain quantifiers.
- It is not always the case that a T -solver for the theory under consideration is available. In this case, a possible solution is to work with a finite axiomatization of T (if it exists), and apply generic first-order theorem proving techniques such as resolution or paramodulation.

Hence, it is necessary to develop techniques and tools that support quantifiers. Although some initial work has already been carried out, we believe there is still a lot of space for improvement.

Challenge 9: For dealing with non-ground formulas, the underlying idea of the existing techniques is based on Herbrand's theorem. That is, the unsatisfiability of a formula is to be detected by generating an unsatisfiable set of ground instances. In order to only generate a small but still sufficient set of instances, one first considers the congruence E generated by all equalities between ground terms in the current partial model. Then, given a non-ground term t occurring in the formula, its relevant ground instances $t\sigma$ are those such that $t\sigma =_E s$ for some ground term $s \in E$.

For given t , s , and E , checking whether such a σ exists is called the *E-matching* problem of t with s . It is well-known to be NP-hard even for fixed s and E : if E is the congruence generated by the 10 ground equations $and(0,0)=0$, $and(0,1)=0$, \dots representing the truth tables of *and*, *or* and *not*, then a propositional formula (a term with variables built over *and*, *or* and *not*) is satisfiable if, and only if, it *E*-matches with 1.

The idea of using *E*-matching for generating a sufficient yet small set of ground instances was first used in the Simplify theorem prover [\[DNS96\]](#) and it has recently been adapted in other SMT solvers such as Yices [\[DdM06a\]](#) or

CVC3 [BT07]. A challenging task, partially studied in [dMB07], is to develop efficient data structures and algorithms that support all necessary operations for E -matching in the context of an SMT solver.

Challenge 10: As already mentioned, the generation of suitable instances is done via E -matching. Since some function and predicate symbols have a predefined semantics given by the theory T , it would be worth considering at least part of this semantics when matching terms. For example, could one use the fact that, when working modulo the theory of linear arithmetic, the function symbol $+$ is associative and commutative and thus generate instances using AC -matching?

Challenge 11: Despite the well-known severe theoretical limitations, it would be interesting to identify fragments and theories for which refutational completeness can be obtained. Even more, by using appropriate redundancy techniques, would it be possible to detect satisfiability in some particular cases?

5.3 SMT for Constraint Programming (CP) and Optimization

In CP (in a broad sense), relations between variables over given domains can be stated in the form of constraints, and the aim is to find values for these variables that satisfy these constraints and/or to optimize some objective function. CP modeling and solving techniques are being applied to problems in a large and broad variety of fields in engineering, (hardware and software) verification, timetabling, traffic and logistics, or finance, among others.

Today it is becoming clearer that SAT and CP techniques share many technological similarities and applications (see the “CP 2006 Workshop on the Integration of SAT and CP techniques”). SAT techniques, when applicable, have the advantage of being very efficient, robust, and highly automatic. On the other hand, the low-level language of propositional logic makes modeling tedious and difficult, and produces non-compact SAT problems, even with extensions such as (weighted) MAX-SAT or pseudo-Boolean constraints that can express optimization. In CP, elegant general formalisms facilitate modeling, and sophisticated special-purpose filtering and propagation algorithms exist for a large diversity of expressive global constraints. But CP implementations are frequently sensitive to variations in the input problem, and tend to need *tuning* by hand to find good heuristics.

Our aim here is to outline several ideas for using SMT, and in particular, our DPLL(T) approach, to combine SAT and CP techniques, hopefully getting the advantages of both and the drawbacks of none.

One lesson most SAT and CP researchers have learned is that techniques that work well on artificial or random problems may not do so on real-world problems, and vice versa³. In the SAT world, this is not surprising, since lemma learning

³ But still, many experiments of CP techniques for real-world applications are being carried out on artificial problems, and problems are sometimes called “non-artificial” because they are translations of, e.g., graph problems which were random or hand-crafted!

is crucial for exploiting the “structure” of real-world problems, a structure that does not exist in random problems. Indeed, on real-world SAT problems, the complete DPLL procedure outperforms incomplete local search methods even for satisfiable problems (see www.satcompetition.org). We now compare complete systematic search methods for SAT and CP on four basic aspects.

Backtracking, backjumping and lemmas

SAT: Conflict analysis techniques allow one to backjump, and at the same time provide the lemmas (in the form of new clauses, i.e., in the same input language!) for preventing similar conflicts in the future.

CP: Techniques for going beyond chronological backtracking exist, as well as notions of lemmas (*nogoods*) for pruning portions of the search space that are known to contain no solutions, but frequently the generality and diversity of the language makes this too difficult. Also the complexity of the constraint filtering and propagation algorithms frequently impedes it, since a notion of *explanation* is required for a precise conflict analysis (as we have seen), and there is no uniform representation language for nogoods in CP, and no uniform conflict analysis and backjump technique (these aspects are highly implementation-dependent).

Heuristics

SAT: One single, robust, general-purpose heuristic is used, based on literal activity (roughly, split on the literal with the highest number of recent occurrences in conflicts and lemmas). One can see this as “working off” *locally* one constraint “cluster” at a time, and, while doing this, extracting lemmas from it, which are kept only while they are *active* (i.e., useful in pruning the search).

CP: Typical heuristics are based on the *first-fail* principle (e.g., *minimum domain*). In practice, tuning is usually needed to find a good heuristic for a given problem, or problem instance. On industrial SAT problems such heuristics behave poorly, consecutively visiting rather unrelated points in the search space, and thus also making it difficult to keep enough useful active lemmas.

Propagation/pruning

SAT: Essentially, only unit propagation is used. Other techniques such as 2-literal-clause reasoning are usually found too expensive.

CP: Sophisticated techniques for propagating and filtering many types of constraints (aimed at important applications) have been developed, maintaining different degrees of (arc, bound, etc.) consistency.

Data structures:

SAT: Refined data structures exist for unit propagation (two-watched literals), clause representation, and bookkeeping for the heuristics.

CP: Again, the generality and diversity of the language makes it hard to develop such data structures. Even for the simple language of propositional CNF, it has taken years of research in SAT solving to reach the current state of the art.

Challenge 12: Develop an SMT system with the advantages of one of CP’s sophisticated global constraint propagation algorithms and the robustness and

efficiency of SAT’s backjumping, lemmas and heuristics. The idea is to express the global constraints as a theory. For instance, we are currently working on the following system.

Example 11. Consider the typical (academic) CP problem of Quasi-Group Completion (QGC, also known as Latin squares). It is important in practice because it appears hidden in many real-world (e.g., scheduling) problems. The question is whether an $n \times n$ table like this one can be completed such that each row and column contains the numbers $1 \dots n$ (in this case $n = 5$):

	3	4		
3	4	5		
4	5			
5				

Currently a good (possibly the best) technique for QGC is the so-called *3-D encoding* into SAT [KRA⁺01], where a propositional variable x_{ijk} means “row i column j has value k ”, and the following clauses are given:

1. *At least one k per $[i, j]$* : clauses like $x_{ij1} \vee \dots \vee x_{ijn}$, and *at most one k per $[i, j]$* : 2-literal clauses like $\neg x_{ij1} \vee \neg x_{ij2}$.
2. The analogous clauses for exactly *one j per $[i, k]$* and *one i per $[j, k]$* .
3. One unit clause per filled-in value, e.g., x_{313} .

With this encoding, in our 5x5 example, DPLL’s UnitPropagate infers no value. But alldifferent constraint filtering on the first three columns and the first row $v_{11}, v_{12}, v_{13}, v_{14}, v_{15}$ reveals that v_{11} and v_{12} consume values 1 and 2 and hence v_{13} must be 3.

Consider an SMT system using this 3-D encoding and where T is the theory of alldifferent. As usual in SMT, the T -solver knows what the x_{ijk} ’s mean. From time to time, one can invoke the T -solver for doing Theory Propagate, but one should apply cheap SAT rules first: UnitPropagate, Backjump, etc. In this case, the T -solver does incremental filtering [Rég94] but must be able to produce explanations. In our example, the theory-propagated literal x_{133} (meaning $v_{13} = 3$) is entailed by $\{ \neg x_{113} \ \neg x_{114} \ \dots \ \neg x_{135} \}$. \square

In this way, the specialized filtering algorithms only need to be extended for generating explanations, but the remaining machinery can be used as it is in $DPLL(T)$: one uniform language (clauses) for expressing no-goods, the conflict analysis mechanism, etc. SAT’s heuristics and unit propagation mechanisms will do what they are good at, which is carrying out the actual search, i.e., the labeling. Learned lemmas help transferring knowledge from the theory to the $DPLL(X)$ engine, which handles it efficiently.

Challenge 13: In the previous example we have considered the alldifferent constraint. Develop explanation-generating T -solvers for other typical global constraints.

Challenge 14: In the previous example, we used a complete underlying encoding into SAT of the QGC problem. Try to exploit the same ideas, but using the boolean part of SMT only for an incomplete encoding.

Challenge 15: In [NO06] we have shown how to model in SMT optimization problems (Max-SAT and Max-SMT) by expressing as an (increasingly stronger) theory T the best solution so far in a branch-and-bound search. How can lower bounds be more effectively applied in that framework?

6 Concluding Remark

We hope that the reader has become challenged and motivated for helping develop this exciting research area and/or for applying SMT techniques and tools.

References

- [BBC⁺05] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: System description: MathSAT 3. In: Nieuwenhuis, R. (ed.) Proceedings of the 20th Conference on Automated Deduction. LNCS (LNAI), vol. 3632, pp. 315–321. Springer, Heidelberg (2005)
- [BD94] Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
- [BT07] Barrett, C., Tinelli, C.: CVC3. In: Computer Aided Verification, 19th International Conference (CAV). Springer LNCS (To appear 2007)
- [DdM06a] Dutertre, B., de Moura, L.: The YICES SMT Solver (2006) <http://yices.csl.sri.com/tool-paper.pdf>
- [DdM06b] Dutertre, B., de Moura, L.M.: 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)
- [DLL62] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Comm. of the ACM* 5(7), 394–397 (1962)
- [dMB07] de Moura, L., Bjorner, N.: Efficient E-matching for SMT Solvers. In: xx, (Submitted 2007)
- [DNS96] Detlefs, D.L., Nelson, G., Saxe, J.: Simplify: the ESC theorem prover. Technical report, Compaq (December 1996)
- [DP60] Davis, M., Putnam, H.: A computing procedure for quantification theory. *Journal of the ACM* 7, 201–215 (1960)
- [DST80] Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpressions problem. *J. of the Association for Computing Machinery* 27(4), 758–771 (1980)
- [ES03] Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT) pp. 502–518 (2003)
- [ILO] ILOG. ilog cplex <http://www.ilog.com/products/cplex>
- [KRA⁺01] Kautz, H.A., Ruan, Y., Achlioptas, D., Gomes, C.P., Selman, B., Stickel, M.E.: Balance and Filtering in Structured Satisfiable Problems. In: Nebel, B. (ed.) 17th International Joint Conference on Artificial Intelligence, IJ-CAI'01, pp. 351–358. Morgan Kaufmann, San Francisco (2001)
- [MMZ⁺01] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. 38th Design Automation Conference (DAC'01) (2001)

- [MSS99] Marques-Silva, J., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48(5), 506–521 (1999)
- [NO05a] Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
- [NO05b] Nieuwenhuis, R., Oliveras, A.: Proof-Producing Congruence Closure. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005)
- [NO06] Nieuwenhuis, R., Oliveras, A.: On sat modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) *SAT 2006*. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
- [NO07] Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. *Information and Computation* 205(4), 557–580 (2007)
- [NOT06] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM* 53(6), 937–977 (2006)
- [Rég94] Régis, J.-C.: A filtering algorithm for constraints of difference in CSPs. In: *Proceedings of 12th National Conference on AI (AAAI'94)*, vol. 1, pp. 362–367, Seattle, (July 31 - August 4, 1994)
- [RT03] Ranise, S., Tinelli, C.: The SMT-LIB Format: An Initial Proposal. In: *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, Miami (2003)
- [RTV97] Roos, C., Terlaky, T., Vial, J.P.: *Theory and Algorithms for Linear Optimization: An Interior Point + Approach*. Wiley, Chichester, UK (1997)
- [SL06] Stump, A., Löchner, B.: Knuth-bendix completion of theories of commuting group endomorphisms. *Inf. Process. Lett.* 98(5), 195–198 (2006)
- [ST05] Stump, A., Tan, L.-Y.: The algebra of equality proofs. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 469–483. Springer, Heidelberg (2005)
- [Ter96] Terlaky, T. (ed.): *Interior Point Methods of Mathematical Programming, Applied Optimization*, vol. 5. Kluwer Academic Publishers, Boston, MA (1996)

On a Logical Foundation for Explicit Substitutions

Frank Pfenning

Department of Computer Science
Carnegie Mellon University
fp@cs.cmu.edu
<http://www.cs.cmu.edu/~fp>

Traditionally, calculi of explicit substitution [1] have been conceived as an implementation technique for β -reduction and studied with the tools of rewriting theory. This computational view has been extremely fruitful (see [2] for a recent survey) and raises the question if there may also be a more abstract underlying logical foundation.

Some forms of explicit substitution have been related to cut in the intuitionistic sequent calculus [3]. While making a connection to logic, the interpretation of explicit substitutions remains primarily computational since they do not have a reflection at the level of propositions, only at the level of proofs.

In recent joint work [4], we have shown how explicit substitutions naturally arise in the study of intuitionistic modal logic. Their logical meaning is embodied by a contextual modality which captures all assumptions a proof of a proposition may rely on. Explicit substitutions mediate between such contexts and therefore, intuitively, between worlds in a Kripke-style interpretation of modal logic.

In this talk we review this basic observation about the logical origin of explicit substitutions and generalize it to a multi-level modal logic. Returning to the computational meaning, we see that explicit substitutions are the key to a λ -calculus where variables, meta-variables, meta-meta-variables, etc. can be unified without the usual paradoxes such as lack of α -conversion. We conclude with some speculation on potential applications of this calculus in logical frameworks or proof assistants.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Lévy, J.J.: Explicit substitutions. *Journal of Functional Programming* 1(4), 375–416 (1991)
2. Kesner, D.: The theory of calculi with explicit substitutions revisited. Unpublished manuscript (October 2006)
3. Herbelin, H.: A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In: Pacholski, L., Tiuryn, J. (eds.) *CSL 1994*. LNCS, vol. 933, pp. 61–75. Springer, Heidelberg (1995)
4. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. *Transactions on Computational Logic (To appear)* (2007)

Intruders with Caps

Siva Anantharaman¹, Paliath Narendran², and Michael Rusinowitch³

¹ Université d'Orléans France

siva@univ-orleans.fr

² University at Albany–SUNY USA

dran@cs.albany.edu

³ Loria-INRIA Lorraine, Nancy France

rusi@loria.fr

Abstract. In the analysis of cryptographic protocols, a *treacherous* set of terms is one from which an intruder can get access to what was intended to be secret, by adding on to the top of a sequence of elements of this set, a *cap* formed of symbols legally part of his/her knowledge. In this paper, we give sufficient conditions on the rewrite system modeling the intruder's abilities, such as using encryption and decryption functions, to ensure that it is decidable if such caps exist. The following classes of intruder systems are studied: linear, dwindling, Δ -strong, and optimally reducing; and depending on the class considered, the cap problem ("find a cap for a given set of terms") is shown respectively to be in P, NP-complete, decidable, and undecidable.

1 Introduction

Cryptography has been applied to render communications secure over an insecure network for many years. However, the underlying difficulties in properly designing cryptographic protocols are reflected by repeated discovery of *logical* bugs in these protocols. As an attempt to solve the problem, there has been a sustained and successful effort to devise formal methods for specifying and verifying the security goals of cryptoprotocols. Various symbolic approaches have been proposed to represent protocols and reason about them, and to attempt to verify security properties such as confidentiality and authenticity, or to discover bugs. Such approaches include process algebra, model-checking, modal logics, equational reasoning, and resolution theorem-proving (e.g., [18,21,8,4]).

In particular, string rewrite systems have provided one of the first formal treatments of security protocol analysis [12], by modeling encryption and decryption as abstract operators. In such a setting, the secrecy property – i.e. whether a message can be deduced by the intruder from observed communications – can be reduced to the so-called *extended word problems*. The approach has been generalized to more realistic protocols by employing term rewrite rules [13,10,17], in particular modeling the capabilities of the intruder in terms of a convergent term rewrite system (*TRS*, for short); more elaborate primitives can be obtained that way. In the analysis of cryptographic protocols using such an approach, the

general cap problem (that we shall define shortly), formally models the possibility that a passive intruder gets hold of a secret m , by using – and possibly re-using – some of the non-public terms that (s)he captures, e.g., by eavesdropping, during a given protocol session. The issue addressed in this paper is how general the (convergent) TRS modeling the intruder’s capabilities can be, so as to get tractable decision procedures for solving this problem.

This paper is structured as follows: In Section 2, after some preliminaries, we formally define the general cap problem, as well as a simpler variant called just the cap problem. Section 3 shows in some detail how these cap problems can be applied to the formal security analysis of protocols. In Section 4, the cap problem is shown to be decidable for optimally reducing string rewrite systems. Section 5 studies the cap problem for (convergent) dwindling TRS R ; it is shown to be decidable in polynomial time; if the TRS R is assumed left-linear in addition, then we show that the set of all irreducible treacherous terms is a regular tree language. Section 6 establishes the undecidability of the cap problem for (convergent) linear, optimally reducing TRS, by reduction from the halting problem for 2-counter machines. In Section 7, we turn our attention to the general cap problem and show that it is NP-complete for special or dwindling TRS. The general cap problem is then studied with respect to rewrite systems R called Δ -strong, more general than dwindling systems; and the decidability of the general cap problem wrt such systems R is shown (Section 8); possible applications are that of modeling homomorphic encryption and the blind signature protocol.

2 Notation and Preliminaries

We assume that the reader is familiar with the well-known notions of terms, rewrite rules and rewrite systems over a given (ranked) signature Σ , and a (possibly infinite) set of variables \mathcal{X} . For any term t , the set of all its positions will be denoted as $Pos(t)$; if $q \in Pos(t)$ then $t|_q$ will denote the subterm of t at position q ; and following Huet [15], the term obtained from t by replacing the subterm $t|_q$ by a term t' will be denoted as $t[q \leftarrow t']$. A similar notation will also be used for the substitution of variables in t by terms. The notions of reduction and of normalization of a term by a rewrite system are assumed familiar too, as well as those of termination and of confluence of the reduction relation defined by such a system on terms. A rewrite system is said to be *convergent* iff the reduction relation it defines on the set of terms is terminating and confluent.

The Cap Problems: Let R be any convergent TRS over some ranked signature Σ and a variable set \mathcal{X} . We assume a ground constant $m \in \Sigma$, referred to as the *secret*, and a subset G of $\Sigma \setminus \{m\}$ referred to as the *intruder repertoire* or *public* symbols. It is assumed that G contains all the root symbols of the left hand sides of all the rules in R and at least one constant, and also that m appears nowhere in the rules of R . (“ R is free from m ”). Symbols which are not in the intruder repertoire are often referred to as *private* symbols. A term that contains only public symbols (and variables) will be said to be a public term; it is said to be private, or non-public, otherwise.

We then extend the signature by adding a set $\{\diamond, \diamond', \diamond'', \dots\}$, of special variables referred to as *hole variables*, or just *holes*; the symbols $\diamond, \diamond_1, \diamond_2, \dots$ (with or without primes) will be used to designate any of the hole variables. A *cap*, or a *cap-term*, is then defined as a public term such that the only variables in it are hole variables. Caps are often represented as $t(\diamond_1, \dots, \diamond_n)$, where the $\diamond_i, 1 \leq i \leq n$, are the distinct hole variables of t , *each of which may occur more than once*. A cap with *exactly one* hole variable occurrence, at a position q , is often more conveniently denoted as $t[\]_q$. The problem referred to in this paper as the *general cap problem*, is the following:

Instance: A convergent TRS R with the properties mentioned above, an intruder repertoire G , and a finite set S of *non-public* ground terms over Σ , at least one of which contains the secret m .

Question: Is there a cap $t(\diamond_1, \dots, \diamond_n)$ over the intruder repertoire G , such that a term $t[\diamond_1 \leftarrow s_1, \dots, \diamond_n \leftarrow s_n]$, with the $s_i \in S$ (not necessarily all distinct), can be R -reduced to m ?

If this question admits a positive answer, the multiset $\{s_1, \dots, s_n\}$, as well as the set S itself, will be said to be *treacherous*, wrt R . The following simpler version of the problem, where the cap has just *one* hole variable occurrence, is referred to as the *cap problem* in the sequel:

Instance': A convergent TRS R with the properties mentioned above, an intruder repertoire G , and a ground term s containing the secret m .

Question': Is there a cap $t[\]_q$ over the intruder repertoire G , such that the term $t[q \leftarrow s]$ reduces to m ?

This simpler version models the possibility that the intruder gets hold of m without re-using any of the intermediary terms captured during a protocol session. The general cap problem will be studied only in the later sections of this paper. We shall first study the (simpler version of the) cap problem, for the following classes of rewrite systems R : string rewrite systems that are either *special* or *optimally reducing*, and term rewrite systems that are either *dwindling* or *optimally reducing*. These notions are formally defined as follows:

- i) R is *special* (or *pure*) iff the rhs of every rule in R is a variable.
- ii) R is *dwindling* iff, for every rule $l \rightarrow r \in R$, r is a *proper* subterm of l .
- iii) R is *optimally reducing* iff, for every $l \rightarrow r \in R$, and for any substitution θ on \mathcal{X} for which $\theta(r)$ is reducible, there is a *proper* subterm s of l such that $\theta(s)$ is reducible¹.

The reason for considering these classes is that, in the formal models of several protocols, term rewrite systems that model the intruder capabilities often belong to these classes. Note that the above three notions are decreasingly restrictive. It is decidable whether a given TRS R is optimally reducing: a non-deterministic polynomial time decision procedure is given in [16].

Recall that a string rewrite system over an alphabet Σ can be seen as the particular case of TRS where the symbols in Σ are all of rank 1. For redactional

¹ This notion was first introduced in [16], and has been extended recently in [9].

reasons, we shall agree to view, in the sequel, any string u over Σ as a term over one variable derived from the *reversed* string of u ; i.e., if $g, h \in \Sigma$ the string gh will be seen as the term $h(g(x))$.

3 Security Analysis and the Cap Problem

String Case: We shall consider first the approach initiated by Dolev and Yao [12] in 1983, for the security analysis of some two-party public key protocols. The protocols are represented as sequences of strings on an alphabet of unary operators. They are defined as insecure if and only if a certain initial message exchanged between the parties can be normalized to the empty string by the intruder rewrite system, by successively adding on caps to the initial message, where each cap is built by using the intruder capabilities. Book and Otto observed, e.g. in [7], that the main technical result behind the Dolev-Yao result can be formulated as follows:

Let R be a convergent special string rewrite system. Then for any regular language L , the set $\{x \mid \exists y \in L : y \rightarrow^* x\}$, of descendants of strings in L , is a regular language. A non-deterministic finite automaton (NFA) accepting this language can be constructed in time polynomial in the total size of R and the size of the NFA.

Term Case: When considering cryptographic protocols defined with operators of arity greater than 1, the extension of cap problems from strings to terms is still relevant for security analysis. Our approach is basically motivated by the logical approach to security (e.g., [18,2,8]).

Consider the following elementary ping-pong protocol introduced in [12]:

$$\begin{aligned} A &\rightarrow B: A, B, \{M\}_{kb} \\ B &\rightarrow A: B, A, \{M\}_{ka} \end{aligned}$$

An intruder impersonating B can mount the following easy attack:

$$\begin{aligned} A &\rightarrow I(B): A, B, \{M\}_{kb} \\ I &\rightarrow B: I, B, \{M\}_{kb} \\ B &\rightarrow I: I, B, \{M\}_{ki} \end{aligned}$$

This can be expressed using encryption and decryption functions e and d respectively and with kb and ki as the keys of B and I respectively. The intruder's initial knowledge includes b, i, kb, ki . At the end of the protocol I gets $e(d(e(m, kb), kb), ki)$. Modulo the convergent term rewrite system \mathcal{I} , consisting of the following rules:

$$d(e(y, u), u) \rightarrow y, \quad e(d(y, u), u) \rightarrow y$$

this is equivalent to $e(m, ki)$. However, this still has not shown that the intruder can get hold of m . For that we have to find a suitable cap for $e(m, ki)$ so that the capped term will normalize to m .

For the case where messages are terms, we shall assume that the intruder has been able to capture some messages from the protocol (we do not study further

how the intruder has interacted with the protocol to get these messages). We will rather focus on the problem of finding caps, and on its complexity, for given intruder knowledge, and given secret. The cap problem is equivalent to what is considered in the literature as a security problem in presence of a *passive intruder*. (It is sometimes referred to as the deduction problem; e.g., [1].)

4 The Cap Problem in the String Case

As we mentioned earlier (Section 3), in the string case the functioning of the protocol is modeled as a regular word grammar, over some given alphabet Σ ; the intruder is assumed active: (s)he is allowed to use the protocol rules for capturing the secret; and the system R modeling the intruder capabilities is assumed convergent. We have then the following extension of the cap problem, the decidability of which is known to be equivalent to deciding protocol insecurity in this case (of an active Dolev-Yao intruder):

Proposition 1. *The following problem is decidable:*

Instance: *An optimally reducing convergent string rewrite system R over an alphabet Σ , an R -irreducible string α , and a regular language $L \subseteq \Sigma^*$.*

Question: *Is there a string $\beta \in L$ such that $\alpha\beta \rightarrow_R^* \lambda$ (the empty string) ?*

Proof. Define $L' = \alpha.L$; then there exists a $\beta \in L$ such that $\alpha.\beta \rightarrow_R^* \lambda$, iff $\lambda \in R^1(L') =$ the set of all R -irreducible descendants of L' . Thus the above proposition can be derived by showing that, for *any* regular language L over Σ , the set $R^1(L)$ of all R -irreducible descendants of L is a regular language too. This is done in the following 3 lemmas.

Lemma 1. *Let R be an optimally reducing convergent string rewrite system over the alphabet Σ . Then every congruence class modulo R is a deterministic context-free language.*

Proof. A deterministic push-down automaton (DPDA) can be constructed for each congruence class. We describe the DPDA in terms of the following transition system on tuples from $\Sigma^* \times \Sigma$. The first component of the tuple has the contents of the stack from bottom to top, and the second component has the current tape symbol. The main loop invariant is that the stack contains an irreducible string – in particular the normal form of the string read so far:

$$\begin{aligned} (w, a) &\mapsto (wa, \epsilon) \text{ if no suffix of } wa \text{ is a redex.} \\ (x'l', a) &\mapsto (xr, \epsilon) \text{ if } l'a \rightarrow r \text{ is a rule} \end{aligned}$$

Checking the condition – whether attaching the tape symbol to the stack contents will create a redex – can be incorporated into the finite control of the DPDA; e.g., by building a trie of all the left-hand sides. \square

Lemma 2. *Let R be an optimally reducing convergent string rewrite system over the alphabet Σ and let $\#$ be a symbol not in Σ . Then the language*

$$\{x\#y \mid x, y \in \Sigma^*, y^{rev} \text{ is } R\text{-irreducible}, x \xrightarrow{!}_R y^{rev}\}$$

is context-free. (y^{rev} is the reverse string of y .)

Proof. The main idea is the same as in the proof of the previous lemma. We construct a DPDA that will scan an input string of the form $x\#y$ from left to right, and will have the normal form of x in the stack when it reaches the tape cell that contains the $\#$ symbol. From then on, the machine pops the stack when the symbol at its top and the tape symbol agree. \square

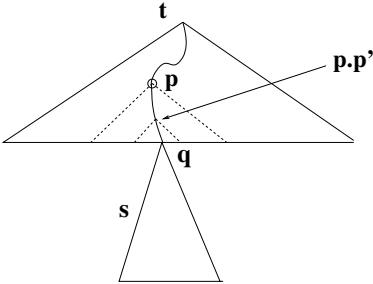
Lemma 3. *Let R be an optimally reducing convergent string rewrite system over the alphabet Σ , and let $L \subseteq \Sigma^*$ be a regular language. Then the language $R^!(L) = \{u \mid \exists v \in L : v \xrightarrow{!}_R u\}$ is a regular language.*

Proof. This follows from the preceding lemma, and the proof is essentially the same as that of Theorem 2.5 in [6]. \square

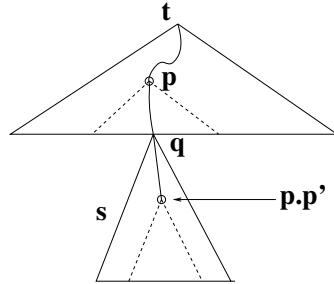
5 Deciding the Cap Problem for Dwindling TRS

We give here a recursive algorithm for solving the cap problem, when the given TRS R is dwindling. With the notation of Section 2, we may assume, without loss of generality, that both the given term s (containing the secret m) and the cap to be found are R -irreducible. Suppose $t|_q$ is a minimal cap (that allows one to deduce the secret m). Let $t' = t[q \leftarrow s]$. Let p be the innermost position where t' is reducible by a rule, say $l \rightarrow r$. Clearly we have $p \prec q$ (for the prefix-ordering ' \preceq ' on positions). Thus $t'|_p = \sigma(l)$ for some substitution σ . Let p' be a position in l such that $l|_{p'} = r$. Thus $\sigma(l)|_{p'} = \sigma(r)$, and $t'|_{p \cdot p'} = \sigma(r)$. Now two cases have to be considered.

case (i) $p \cdot p' \preceq q$: In this case $t'|_p$ reduces to $t'|_{p \cdot p'}$. Hence $t[p \leftarrow t|_{p \cdot p'}]_q$ will be a smaller cap (with a hole at some position above q), so this case need not be considered. The sub-case where $p \cdot p' = q$ for every possible redex is one where there is no cap, and one exits with failure.



Case (i)



Case (ii)

The case where $p \cdot p'$ and q are incomparable need not be considered since the cap is irreducible (and cannot contain any occurrence of m); so we go to:

case (ii) $q \prec p \cdot p'$: Let $q = p \cdot q'$ and $p' = q' \cdot q''$. We have here: $q' \prec p'$, $\sigma(l|_{q'}) = s$. And $\sigma(l)$ reduces to $s|_{q''}$ which is a proper subterm of s . Thus s is treacherous if and only if so is $s|_{q''}$.

We thus derive a procedure for checking whether a term is treacherous:

1. If $s = m$ RETURN **true**;
 else non-deterministically choose a rule $l \rightarrow r$ and a proper subterm l' of l that is a *proper* superterm of r ; let $l' = l|_q$.
2. Let $\theta = mgu(s =? l')$.
3. If $\theta(l[q \leftarrow \diamond])$ has symbols which are not in the intruder repertoire,
 or if $\theta(r)$ does *not* contain m , then **fail**;
 else set $s := \theta(r)$ and GOTO 1.

This can also be done in a bottom-up (dynamic programming-like) way. Clearly m itself is treacherous. Now suppose all proper subterms of s have been tested for treachery and the results recorded. Testing whether s itself is treacherous can then be done by modifying Step 3 above to:

- 3bis. If $\theta(l[q \leftarrow \diamond])$ has symbols which are not in the intruder repertoire, **fail**;
 else check whether $s := \theta(r)$ is treacherous: if **yes** RETURN **true** else **fail**.

Making this deterministic requires that each such l' be tried in Step 1. This could take $O(|R||s|)$ time in the worst case, where $|R|$ is the total size of the term rewrite system. Thus the total complexity is $O(|R||s|^2)$.

5.1 Case of Left-Linear Dwindling TRS: A Regularity Result

Proposition 2. *Let R be a left-linear, dwindling and convergent TRS. Then the set of all irreducible treacherous terms, wrt R , is a regular tree language.*

Before proving this proposition, let us observe that the hypothesis of *left-linearity* is essential, as the following example shows: consider the special TRS formed of the unique rule: $f(g(x, x, y)) \rightarrow y$, where f is public; then, clearly $g(t_1, t_2, m)$ is treacherous if and only if $t_1 = t_2$. Note also that the hypothesis of *irreducibility* is also needed, as is seen with the example of the string rewrite system with a single rule $fg \rightarrow \lambda$; the set of *all* treacherous terms here is non-regular, since its intersection with f^*g^* is the language $\{f^n g^m \mid n \geq m\}$; but the set of all *irreducible* treacherous terms is $\{f^n \mid n \geq 0\} = f^*$.

The above proposition is proved via the following lines of reasoning:

- (i) we construct a regular tree grammar \mathcal{G} that generates a subset of the set of all treacherous terms wrt R , which includes all irreducible treacherous terms;
- (ii) since R is assumed left-linear, the set $IRR(R)$ of all ground terms in R -normal form is known to be a regular tree language (cf. e.g., [14]);
- (iii) The set of all irreducible treacherous terms is then obtained as the intersection of the language of \mathcal{G} with $IRR(R)$.

The construction of the tree grammar is based on the fact that, when the TRS R is (dwindling, and) also left-linear, we can reformulate the algorithm of the previous section more precisely, as follows:

- 1'. Non-deterministically choose a rule $l \rightarrow r$ and a *proper* subterm $l' = l|_q$ of l , that is a *proper* superterm of r , with the additional property that $(l[q \leftarrow \diamond])$ has all its symbols inside the intruder repertoire.

- 2'. Let $\theta = mgu(s =? l')$.
 3'. Set $s := \theta(r)$, and GOTO 1'.

See [3] for the details of the construction of the tree grammar.

6 Undecidability Results

Unfortunately the cap problem is undecidable even for optimally reducing, linear, and convergent TRS: we prove that there is a *fixed* optimally reducing, linear, and convergent TRS for which the cap problem is undecidable. The proof is by reduction from the halting problem for 2-counter Minsky machines. A *configuration* of such a 2-counter machine, at any given stage of a computation, will be seen as a triple (C_1, q_l, C_2) where l is the (label of the) next instruction to execute, and $C_i, i = 1, 2$, are the current values of the two counters.

The halting problem for any program \mathcal{P} of such a machine will be encoded as a cap problem over an optimally reducing TRS, which is linear and convergent. For doing that, any state symbol q_l will be seen as a unary function symbol. In addition we introduce further function symbols f, s of rank 1 and c of rank 3, and constants $m, 0$. The constant m stands for some secret message, 0 encodes the natural integer zero, s encodes the successor function on integers, and c encodes configuration triples. The following rules do the encoding (where l, l' stand for suitable instruction labels, 0 (resp. L) being the label for start (resp. halt)):

Initial and final configurations (resp. with given k, p , and k', p'):

$$c(s^k(0), q_0(m), s^p(0)), \quad c(s^{k'}(0), q_L(m), s^{p'}(0))$$

Incrementation of counter 1 or 2:

$$f(c(x, q_l(z), y)) \rightarrow c(s(x), q_{l+1}(z), y), \quad f(c(x, q_l(z), y)) \rightarrow c(x, q_{l+1}(z), s(y))$$

Conditional decrementation of counter 1 or 2:

$$f(c(s(x), q_l(z), y)) \rightarrow c(x, q_{l+1}(z), y), \quad f(0, q_l(z), y) \rightarrow c(0, q_{l'}(z), y). \\ f(c(x, q_l(z), s(y))) \rightarrow c(x, q_{l+1}(z), y), \quad f(x, q_l(z), 0) \rightarrow c(x, q_{l'}(z), 0).$$

At STOP, release the secret m : $f(c(s^{k'}(0), q_L(z), s^{p'}(0))) \rightarrow z$.

The role played by f is to ensure that this rewrite system is terminating. The cap problem over this rewrite system – which is obviously linear and optimally reducing – with $\{0, f, q_1, \dots, q_N\}$ as the intruder repertoire, obviously encodes the halting problem for the 2-counter machine programs. We deduce that the cap problem is undecidable even for linear and optimally reducing systems.

7 The General Cap Problem

The definition of the notion of cap with one hole, as given in Section 2, does not allow the intruder to re-use terms. For instance, consider the rewrite system R with a single rule:

$$g(f(x, a), f(y, a)) \rightarrow x,$$

where g is in the intruder repertoire, but f and a are not. For the definition of cap with one hole, $f(m, a)$ is not treacherous. But if $f(m, a)$ can be re-used

then m can be recovered since $g(f(m, a), f(m, a))$ reduces to m . In other words, there is a (non-linear) cap $t(\diamond) = g(\diamond, \diamond)$ such that $t[\diamond \leftarrow f(m, a)] \rightarrow_R^! m$. Also, there could be more than one term containing m that the intruder may be able to use; this explains that the general cap problem allows more general contexts, with more than one hole.

We show now that the general cap problem is NP-complete for dwindling (and convergent) TRSs.

Proposition 3. *The general cap problem is NP-hard, even for special TRS.*

Proof. The proof is by reduction from the 3-colorability problem. Let (V, E) be any arbitrarily given undirected graph. Introduce a function symbol g of rank $|E| + 1$, a symbol f of rank 2, and a symbol h of rank 1. Associate a variable x_j to each node v_j in V , and represent every edge $e_i = (v_j, v_k)$ in E (joining the two nodes v_j, v_k in the graph) by the term $t_i = f(x_j, x_k)$; finally let B, G and R be constants that correspond to the 3 colors. We then consider the pure TRS formed of the following single rule:

$$g(t_1, \dots, t_{|E|}, h(u)) \rightarrow u$$

where u is a new variable, not appearing in any of the terms t_i . Assume that g is the only symbol in the intruder repertoire, i.e., all symbols other than g are private. Let $f(B, R), f(R, B), f(G, R), f(R, G), f(G, B), f(B, G)$ and $h(m)$ be the terms known to the intruder. Then it is not hard to see that m can be obtained by the intruder (by plugging in a treacherous set of terms in $g(\diamond_1, \dots, \diamond_{(|E|+1)})$), if and only if the graph can be colored with the 3 colors B, R, G . \square

We shall be showing below that the general cap problem is in NP for dwindling TRS. A few preliminaries are needed for proving that.

7.1 The \mathcal{I} -closure of a Set of Terms

Given a finite set S of private ground terms, we define the set of \mathcal{I} -constructible terms, referred to as the \mathcal{I} -closure $\mathcal{I}(S)$ of S , as the smallest set such that: (i) $S \subseteq \mathcal{I}(S)$, (ii) If $f^{(p)}$ is a public function symbol and s_1, \dots, s_p are \mathcal{I} -constructible terms, then $f(s_1, \dots, s_p) \in \mathcal{I}(S)$, and (iii) Nothing else is in $\mathcal{I}(S)$.

(The \mathcal{I} refers to the intruder repertoire.) It is not hard to see that $\mathcal{I}(S)$ is a regular tree language for any given finite set S (see e.g., the proof of Proposition 7 below). Define a set of terms $\Gamma = \{t_1, \dots, t_n\}$ to be \mathcal{I} -independent if and only if for all t_i , we have $t_i \notin \mathcal{I}(\Gamma \setminus \{t_i\})$; it is easy to see then, that from every finite set S of terms we can extract an \mathcal{I} -independent subset, with the same \mathcal{I} -closure. A ground substitution θ is \mathcal{I} -independent if and only if $\text{Ran}(\theta)$ is an \mathcal{I} -independent set, and $\forall v_i, v_j \in \text{Dom}(\theta) : \theta(v_i) = \theta(v_j) \Leftrightarrow v_i = v_j$.

From the definitions, we directly get the following: If S is an \mathcal{I} -independent set of terms, then a term s is in $\mathcal{I}(S)$ if and only if there is a cap $t(\diamond_1, \dots, \diamond_n)$ and an \mathcal{I} -independent substitution $\theta = [\diamond_1 \leftarrow s_1, \dots, \diamond_n \leftarrow s_n]$, with $s_i \in S$ for all i , such that $s = \theta(t)$. The following property is easily established too:

Proposition 4. *Let S be a treacherous set of terms and let $t(\diamond_1, \dots, \diamond_n)$ be a cap for S such that a term $t[\diamond_1 \leftarrow s_1, \dots, \diamond_n \leftarrow s_n]$, with the $s_i \in S$, can be R -reduced to m . If S' is an \mathcal{I} -independent subset of S with $\mathcal{I}(S) = \mathcal{I}(S')$, then there is a cap $t'(\diamond'_1, \dots, \diamond'_k)$, and an \mathcal{I} -independent substitution θ with $\text{Range}(\theta) \subseteq S'$ such that $\theta(t') = t[\diamond_1 \leftarrow s_1, \dots, \diamond_n \leftarrow s_n]$.*

As a corollary, we deduce that every set of treacherous terms has an \mathcal{I} -independent treacherous subset. The proofs of the following two propositions can be found in [3].

Proposition 5. *Let $\theta = [x_1 \leftarrow s_1, \dots, x_n \leftarrow s_n]$ be an \mathcal{I} -independent substitution assigning non-public terms s_i to variables x_i , $1 \leq i \leq n$. Then θ unifies two public terms t_1 and t_2 if and only if $t_1 = t_2$.*

Proposition 6. *Let $l \rightarrow r$ be a rewrite rule, s be a term such that $\text{Pos}(l) \subseteq \text{Pos}(s)$ and θ be a ground substitution such that $\theta(s)$ is an instance of l . Then either s is an instance of l or there are distinct subterms s_1 and s_2 of s such that $\theta(s_1) = \theta(s_2)$.*

Corollary 1. *Let $l \rightarrow r$ be a rewrite rule, s be a public term such that $\text{Pos}(l) \subseteq \text{Pos}(s)$ and θ be an \mathcal{I} -independent ground substitution such that $\theta(s)$ is an instance of l . Then s is an instance of l .*

Proof. By Proposition 6 there must be distinct subterms s_1 and s_2 of s such that θ is a unifier of s_1 and s_2 . But by Proposition 5, s_1 and s_2 must be identical which is a contradiction. \square

Proposition 7. *Let S be an \mathcal{I} -independent set of ground terms and t any given term. Then the problem of checking whether t has an instance in $\mathcal{I}(S)$ is in $\text{NTIME}(|t| + |S|)$ where $|S| = \text{sum of the sizes of terms in } S$.*

Proof. We represent the given set S of ground terms as a (not necessarily rooted) dag $G = (V, E)$; let $V = \{n_1, \dots, n_l\}$. We define the following mapping with each node in V :

$$\begin{aligned} \nu(n_i) &= (n_i, 1) \text{ if the term at } n_i \text{ is in } S, \\ &= (n_i, 0) \text{ otherwise.} \end{aligned}$$

Each such pair (n_i, b) will be seen as a state of a tree automaton A ; we also add a distinguished state q_{acc} , which will be the only accepting state of A . For all nodes n_j , if $f^{(l)}$ is the symbol at the node and n_{j_1}, \dots, n_{j_l} are the nodes corresponding to the ordered arguments of f (not necessarily distinct), then we form a transition rule of A :

$$f(\nu(n_{j_1}), \dots, \nu(n_{j_l})) \rightarrow \nu(n_j).$$

If $\nu(n_j) = (n_j, 1)$, then we also form the rule:

$$f(\nu(n_{j_1}), \dots, \nu(n_{j_l})) \rightarrow q_{acc}.$$

Finally, for all public symbols g , we add the transition rule:

$$g(q_{acc}, \dots, q_{acc}) \rightarrow q_{acc}.$$

The size of the automaton A is obviously linear in $|S|$. The automaton is non-deterministic, but it is easily checked that every term in $\mathcal{I}(S)$ has a unique accepting run. Now consider the problem of checking whether a given term t has an instance in $\mathcal{I}(S)$. If p_1, \dots, p_n are the variable positions of t , then we guess the states at each position, say q_1, \dots, q_n respectively;

1. we have then to verify that this state assignment can be completed into an accepting run for t ; *and*

2. for each variable $x \in \mathcal{Var}(t)$, if p_{x_1}, \dots, p_{x_j} are the positions where it occurs and q_{x_1}, \dots, q_{x_j} the corresponding states, then check whether there is a common term t' that all these states “inhabit” — i.e., each state appears at the root of a run of A on the term t' .

Checking this latter requirement (although EXPTIME-hard, in general) is very easy in our case: the only way that q_{x_1}, \dots, q_{x_j} can appear at the roots of runs for the same term, is if one of the following holds:

(a) they are all the same, or

(b) $\{q_{x_1}, \dots, q_{x_j}\} = \{q_{acc}, (n, 1)\}$ for some $n \in V$. \square

Another way of stating the above conditions (a) and (b), is as follows: for each variable $x \in \mathcal{Var}(t)$, if p_{x_1}, \dots, p_{x_j} are the positions where it occurs and q_{x_1}, \dots, q_{x_j} the corresponding states, then $\{q_{x_1}, \dots, q_{x_j}\}$ is:

(a') either $\{q_{acc}\}$;

(b') or $\{\nu(n_i)\}$ for some $n_i \in V$;

(c') or $\{q_{acc}, (n, 1)\}$ for some $n \in V$.

This enables us to formulate the following NP-algorithm: for each variable that occurs more than once, guess which of the conditions (a'), (b') or (c') will hold. If (a') then replace x with a public constant c ; if (b') or (c') then replace x with the term corresponding to the node. Finally, a linear term s has an instance in $\mathcal{I}(S)$ if and only if s matches with a term in S , or $s = f(s_1, \dots, s_m)$, where f public, and each $s_i, 1 \leq i \leq m$, has an instance in $\mathcal{I}(S)$.

The deterministic version of this algorithm (i.e., exhaustive search instead of guessing) has time complexity $O(3^k (|t| + |S|))$ where k is the number of variables that occur more than once. Thus we have a polynomial time algorithm, *for the case where the number of variables that occur more than once is fixed in advance.*

Proposition 8. *Let S be an \mathcal{I} -independent set of ground terms, and t any given term. Then checking whether t has an instance in $\mathcal{I}(S)$ can be done in time $O(3^k (|t| + |S|))$, where $k =$ number of variables occurring more than once in t .*

7.2 A Procedure for the General Cap Problem

We propose an inference rule and a saturation procedure in order to derive the secret m from a given set S of non-public terms. The procedure can be shown to terminate for all convergent term rewrite systems. It is not complete in general; however, completeness can be shown for dwindling, and Δ -strong TRSs. And in the dwindling case, the algorithm will be shown to run in NP time.

Let $\mathcal{FPos}(t)$ be the set of non-variable positions in any term t : $\mathcal{FPos}(t) = \{p \mid p \in \mathcal{Pos}(t), t|_p \text{ is not a variable}\}$. In the proof details below, we shall be denoting by ‘ \preceq ’ the subterm ordering on terms, as well as the prefix ordering on the positions on a term, interchangeably. The inference rule is as follows:

$$\frac{S \uplus \{s\}}{S \cup \{s, \sigma(r)\}} \quad (l, p) \quad \text{where } (l \rightarrow r) \in R, \sigma = mgu(s =^? l|_p), p \in \mathcal{FPos}(l), \\ \sigma(r) \prec s, \text{ and } \sigma(l) \text{ has an instance in } \mathcal{I}(S).$$

The set S is said to be *saturated* iff it doesn’t grow under any application of this inference rule. (Clearly S is treacherous if the saturated set contains m .)

Lemma 4. *Any set of private terms S can be saturated in finitely many steps.*

Proof. One way to see this is to view the inference step as an ordered rewriting step using the rewrite rules $R' = \{(l|_p \rightarrow r) \mid (l \rightarrow r) \in R, p \in \mathcal{FPos}(l)\}$. Each term has only finitely many descendants modulo R' , so the saturation process cannot lead to an infinite set of terms. \square

The saturation procedure given above is incomplete for general TRS and arbitrarily chosen simplification orderings \succ ; cf. [3].

7.3 An NP-Decision Procedure for dwindling TRS

Proposition 9. *The general cap problem is in NP for any dwindling (convergent) term rewrite system R .*

Proof. The proof uses the following two lemmas (notation of Section 2):

Lemma 5. *Let R be a (convergent) dwindling TRS and let S be a saturated set of private terms. Then S is treacherous if and only if $m \in S$.*

Proof. Assume the contrary. Let S be a saturated set of private terms that is treacherous but does not contain m . Let t' be a \succ -minimal term in $\mathcal{I}(S)$ whose irreducible normal form is m . By Proposition 4 there must be a cap $t(\diamond_1, \dots, \diamond_k)$ and an \mathcal{I} -independent substitution $\theta = [\diamond_1 \leftarrow s_1, \dots, \diamond_k \leftarrow s_k]$, whose range is a subset of S , such that $t' = \theta(t)$. Suppose t' is reducible by a rule $l \rightarrow r$, and $t'|_p = \sigma(l)$ for some substitution σ . Let p_1, \dots, p_n be the variable positions of l , and let $\pi_i = p \cdot p_i$ for $i = 1, \dots, n$. Now $l \rightarrow r$ is a dwindling rule, i.e., $r = l|_{p'}$ for some position p' ; there are two cases to be considered.

- (i) $p \cdot p'$ is a position in t . Then $t[p \leftarrow t|_{p \cdot p'}]$ is in $\mathcal{I}(S)$, which contradicts the minimality of t' .
- (ii) There is a variable \diamond_i at some position q_i in t such that $q_i \prec p \cdot p'$; hence s_i unifies with a subterm of l . (The situation is like in case (ii), Section 5.)
Now, since S is assumed to be saturated, $\sigma(r)$ has to be already in S . Thus $t'[p \leftarrow \sigma(r)]$ is in $\mathcal{I}(S)$ too, which is a contradiction. \square

In the light of the above proof, we can modify the inference rule for the dwindling case to:

$$\frac{S \uplus \{s\}}{S \cup \{s, \sigma(r)\}} \quad (l, p) \quad \text{where } (l \rightarrow r) \in R, \sigma = \text{mgu}(s = ? l|_p), p \in \mathcal{FPos}(l), \\ r \text{ is a proper subterm of } l|_p, \text{ and } \sigma(l) \text{ has an instance} \\ \text{in } \mathcal{I}(S).$$

Let $\|R\| = \sum_{(l_i \rightarrow r_i) \in R} |l_i|$. We then have:

Lemma 6. *The saturation of any given set of private terms S can be done in $\text{NTIME}(\phi(|S|, \|R\|))$, where ϕ is a polynomial with two arguments.*

Proof. Every term added under an inference is a proper subterm of some term in S ; by Proposition 7 each inference step can be performed in NP time. \square

If the number of variables that occur more than once in the lhs of the rules in R is *fixed*, we get a polynomial-time algorithm by Proposition 8; and this gives:

Proposition 10. *Let k be a fixed natural integer, and R a dwindling TRS such that, for each $l \rightarrow r \in R$ the number of variables in $\text{Var}(l) \setminus \text{Var}(r)$ that occur more than once in l is less than k . Then, the general cap problem over R , and a given set of private terms S , is decidable in polynomial time over $\|R\|$ and $|S|$.*

8 Δ -Strong Intruder Theories

Let R_0 be any given convergent intruder TRS. An n -ary public symbol f is said to be *transparent* for R_0 , or *R_0 -transparent*, if and only if, for all x_1, \dots, x_n , there exist cap-terms $t_1(\diamond), \dots, t_n(\diamond)$ such that $t_i[\diamond \leftarrow f(x_1, \dots, x_n)] \rightarrow_{R_0}^* x_i$, for every $1 \leq i \leq n$. For instance, the public function p (“pair”) is transparent for the TRS: $\pi_1(p(x, y)) \rightarrow x$, $\pi_2(p(x, y)) \rightarrow y$, where π_1 and π_2 are public.

It is clear that if the general cap problem is decidable for R_0 , then so is checking R_0 -transparency. We shall consider public constants to be transparent for any intruder system R_0 . A public function symbol is *R_0 -resistant* iff it is not R_0 -transparent. Private functions will be considered to be resistant, for any intruder system R_0 . By definition, an R_0 -resistant term is one whose top-symbol is R_0 -resistant.

Let R be any convergent intruder TRS, and \succ a simplification ordering containing R . (Note: the notation ‘ \succ ’ for the term ordering should not cause any confusion with the prefix-ordering for the positions on terms, since ‘ \succ ’ is a simplification ordering.) We assume that \succ satisfies the *block-ordering property*: every private symbol is higher than every public symbol under \succ . We shall denote by Δ a subsystem consisting of (some of the) dwindling rules in R . A rewrite rule $l \rightarrow r$ is said to be Δ -strong, wrt the simplification ordering \succ , if and only if every Δ -resistant subterm of l is greater than r wrt \succ . The intruder TRS R is said to be Δ -strong wrt \succ if and only if every rule in $R \setminus \Delta$ is Δ -strong wrt \succ .

Lemma 7. *Let R be a convergent intruder TRS, Δ a convergent dwindling subsystem of R , and suppose R is Δ -strong wrt a simplification ordering \succ , total on ground terms, that contains R . Then, any set S of private terms that is saturated (for the inference rule of Section 7.2), is R -treacherous if and only if $m \in S$.*

The proof is very similar to that of Lemma 5. The details are not given here for want of space; see 3. We thus derive:

Proposition 11. *The following problem is decidable:*

Instance: *A convergent TRS R over the intruder repertoire, Δ a dwindling, convergent subsystem of R , a simplification ordering \succ wrt which R is Δ -strong, a free constant m , and a finite set S of irreducible non-public ground terms, at least one of which contains m .*

Question: *Is there a cap-term $t(\diamond_1, \dots, \diamond_k)$ such that $t[\diamond_1 \leftarrow s_1, \dots, \diamond_k \leftarrow s_k]$, with the $s_i \in S$ (not necessarily all distinct), can be R -reduced to m ?*

Applications. (i) One can handle the cap problem for homomorphic encryption (i.e., ‘encryption’ e distributes over ‘pair’), by the Δ -strong approach, with the following convergent TRS R ; the rules to the left form Δ ; d, e are Δ -resistant:

$$\begin{array}{ll} \pi_1(\text{pair}(x, y)) \rightarrow x & d(e(x, y), y) \rightarrow x \\ \pi_2(\text{pair}(x, y)) \rightarrow y & e(d(x, y), y) \rightarrow x \\ & e(\text{pair}(x, y), z) \rightarrow \text{pair}(e(x, z), e(y, z)) \\ & d(\text{pair}(x, y), z) \rightarrow \text{pair}(d(x, z), d(y, z)) \end{array}$$

Related results were obtained in 10, with a more complex proof (but with a polynomial time algorithm).

Homomorphic encryption and signature have several applications, such as e-voting, auction, and private information retrieval.

(ii) The blind signature protocol can be modeled by rewrite rules of the form $\mathcal{U}(S_A(\mathcal{B}_A(x, y)), y) \rightarrow S_A(x)$, where \mathcal{B}_A is the blinding function (of B wrt to signer A), S_A is the signing function of A , and \mathcal{U} is the unblinding function. Such systems are covered by the Δ -strong approach, by setting $\mathcal{B}_A \succ S_A$, for every signer A . One can also handle Block-Cipher related theories, such as the one obtained by adding the rule $\text{split}(e(\text{pair}(x, y), z)) \rightarrow e(x, z)$ to the dwindling system Δ of the previous example (i).

Remark. No polynomial upper bound can be given for the number of terms added to S , under saturation, in the Δ -strong case (unlike in Lemma 6 for the dwindling case). It is possible to show that the general cap problem for Δ -strong intruder theories is PSPACE-hard, by a reduction from the membership problem for labeled bounded automata (LBA).

9 Related Works, Conclusion

In 11, the authors have studied intruder theories given by convergent public-collapsing systems. They give an NP-decision procedure for protocol insecurity in the case of an active intruder. In 11, the authors present an algorithm for the general cap problem for an intruder given by a convergent *dwindling* rewrite system. They in fact considered the more general *static equivalence* problem, and their algorithm was proved to be polynomial when the *size* of the rewrite

system is *fixed*. This work has been extended by [5] to the insecurity problem for active intruders; but the author does not give any complexity result.

We have shown that the extension of even the (single hole) cap problem to the slightly more general class of optimally reducing, (convergent) and linear TRS leads to undecidability; but our decidability result for such systems in the string rewriting case improves upon the results of Book and Otto [7]. The NP-complexity bound established for the general cap problem wrt dwindling TRS, can be seen as adding precision to some results of [1]; actually, our complexity result of Proposition 10, for the general cap problem over such systems, is stronger than the corresponding result of [1]. We have also given an algorithm for the general cap problem for a class of intruder theories not considered in [11,15], namely the Δ -*strong* one, thereby deriving a new security result for passive intruders. In [10], a deduction problem analogous to the general cap problem is investigated, by using specific deduction rules for encryption and pairs (unlike ours), and it is unclear how the results can be compared.

The decidability results derived in this paper cover several theories of interest for security protocols. It would be of interest to extend them to AC-rewrite systems, in order to capture important theories comprising AC-operators (e.g., abelian groups). It would be important too, to try to lift our results to cover the case of active intruders, by integrating constraint solving and semantic unification algorithms.

References

1. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. *Theor. Comput. Sci.* 367(1-2), 2–32 (2006)
2. Amadio, R., Lugiez, D., Vanackère, V.: On the symbolic reduction of processes with cryptographic functions. *Theor. Comput. Sci.* 290(1), 695–740 (2003)
3. Anantharaman, S., Narendran, P., Rusinowitch, M.: Intruders with Caps (2007) <http://www.univ-orleans.fr/lifo/prodsci/rapports/RR/RR2007/RR-2207-02.ps>
4. Armando, A., Compagna, L.: SATMC: a SAT-based Model Checker for Security Protocols. In: Alferes, J.J., Leite, J.A. (eds.) *JELIA 2004*. LNCS (LNAI), vol. 3229, pp. 730–733. Springer, Heidelberg (2004)
5. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: *Proc. of ACM Conference on Computer and Communications Security*, pp. 16–25 (2005)
6. Book, R.V., Jantzen, M., Wrathall, C.: Monadic thue systems. *Theor. Comput. Sci.* 19, 231–251 (1982)
7. Book, R.V., Otto, F.: The verifiability of two-party protocols. In: *EUROCRYPT*, pp. 254–260 (1985)
8. Chevalier, Y., Vigneron, L.: A Tool for Lazy Verification of Security Protocols. In: *Proc. of the Automated Software Engineering Conference (ASE'01)*, IEEE Computer Society Press, Washington, DC, USA (2001)
9. 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, pp. 294–307. Springer, Heidelberg (2005)

10. Comon-Lundh, H., Treinen, R.: Easy Intruder Deductions. In: Dershowitz, N. (ed.) *Verification: Theory and Practice*. LNCS, vol. 2772, pp. 225–242. Springer, Heidelberg (2004)
11. Delaune, S., Jacquemard, F.: A decision procedure for the verification of security protocols with explicit destructors. In: *Proc. of ACM Conference on Computer and Communications Security*, pp. 278–287 (2004)
12. Dolev, D., Yao, A.C.: On the security of public key protocols. *IEEE Transactions on Information Theory* 29(2), 198–207 (1983)
13. Durgin, N.A., Lincoln, P.D., Mitchell, J.G., Scedrov, A.: Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security* 12(1), 677–722 (2004)
14. Gilleron, R., Tison, S.: Regular tree languages and rewrite systems. *Fundamenta Informaticae* 24, 157–176 (1995)
15. Huet, G.P.: Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM* 27(4), 797–821 (1980)
16. Narendran, P., Pfenning, F., Statman, R.: On the unification problem for Cartesian Closed Categories. *Journal of Symbolic Logic* 62(2), 636–647 (1997)
17. Nesi, M., Rucci, G.: Formalizing and analyzing the Needham-Schroeder symmetric-key protocol by rewriting. *Electr. Notes Theor. Comput. Sci.* 135(1), 95–114 (2005)
18. Weidenbach, C.: Towards an automatic analysis of security protocols. In: Ganzinger, H. (ed.) *Automated Deduction - CADE-16*. LNCS (LNAI), vol. 1632, pp. 378–382. Springer, Heidelberg (1999)

Tom: Piggybacking Rewriting on Java

Emilie Balland¹, Paul Brauner¹, Radu Kopetz², Pierre-Etienne Moreau²,
and Antoine Reilles³

¹ UHP & Loria

² INRIA & Loria

³ INPL & Loria

Abstract. We present the TOM language that extends JAVA with the purpose of providing high level constructs inspired by the rewriting community. TOM furnishes a bridge between a general purpose language and higher level specifications that use rewriting. This approach was motivated by the promotion of rewriting techniques and their integration in large scale applications. Powerful matching capabilities along with a rich strategy language are among TOM's strong points, making it easy to use and competitive with other rule based languages.

1 Introduction

Term Rewriting provides a theoretical framework that is very useful to model, study, and analyze various parts of a complex system, from algorithms to running software. During the last 20 years, there were many successful attempts in understanding, certifying, and proving properties of software, such as termination or confluence.

Term Rewriting is also a great tool for building software. Following the development of Lisp, the first equational interpreters, OBJ and EQI were introduced in 1975 by J. A. GOGUEN and M. J. O'DONNELL. Many tools have integrated the notion of term rewriting in their implementation, among them, let us mention Reve 1984, ML 1985, Clean 1986, RRL 1988, ASF+SDF 1989, Spike 1992, ELAN 1993, Larch Prover 1993, Caml 1993, Otter 1994, MAUDE 1995, CafeOBJ 1995, CiME 1996, DMS 1997, Stratego 1998, Hats 1999, TOM 2001, *etc.* Some of them are not only tools which use the notion of term rewriting, but general purpose programming languages whose semantics and execution mechanism are fully based on the notions invented, defined, and studied by the rewriting community: *term, pattern matching, equational theory, rewrite rule, strategy, etc.* to mention just a few of them.

Throughout the course of 10 years we developed the ELAN system [3]. We integrated some of the best algorithms to implement pattern-matching, rule based normalization, and non-deterministic computations. As a result, ELAN is certainly, along with MAUDE [8], one of the most established term rewriting based implementations which efficiently compiles associative-commutative rewriting combined with non-deterministic strategies. This was a very fruitful experience and it taught us many lessons. One of them is that implementing

a good term data-structure is difficult, data conversion (also known as *marshalling*) is often a bottleneck, integrating built-in data-types such as integers or doubles is not so easy, mutable data-structures, such as arrays, are essential for the implementation of efficient data-structures like hash-tables. But one of the most important things we learned is that even if *efficiency* is important to make our technology credible, *integration capabilities* are even more important to make our research widely used both in academic and industrial projects.

In 2001 we started the design of a new language called TOM [12], available at <http://tom.loria.fr/>, whose goal was to pursue the promotion of term rewriting by making our concepts and techniques more easily available. One solution could have been to increase our implementation efforts by providing more data-structures, more libraries, more input/output facilities, more threads, more native interfaces, more graphical user interfaces, more, more, more. But we have to admit that these are difficult tasks and that many other languages already do that very well. In fact, this is not our main business. Therefore, we chose another approach: make our technology available on top of an existing language. This concept is called *Formal Island* [1].

In a first possible scenario, we start from an already existing application and our goal is to implement new functionalities, or re-implement some old ones, using rewriting. The expected result is a more concise and abstract description, and the possibility to reason about this new piece of code. In this case, the data-structure used by the application are already defined. We cannot translate them forth and back before and after each rewrite step, this would introduce unacceptable marshalling costs. Behind the notion of *formal island* there is the notion of *formal anchor*, also called *mapping*, which describes how a concrete data-structure can be seen as an algebraic term. This idea, related to P. WADLER's views, allows TOM to rewrite any kind of data structure, as long as a *formal anchor* is provided. In a second scenario, the application is both specified and implemented at the same time, using rewrite rules. In that case, the system should be easy to use: the definition of an algebraic signature, the definition of rules, and that's all! In this paper we focus on this second approach, which is exactly what TOM provides when the underlying host language is JAVA.

The paper is organized as follows. In Section 2, we present how term rewriting is implemented and integrated into JAVA. Section 3 focuses on the strategy language and its control mechanism. Section 4 exposes meta-programming features added to the strategy language for managing non-deterministic computations and for modifying strategies at runtime. The two following sections present respectively some key details of the TOM implementation and some significant applications. Section 7 and 8 present related work and conclude.

2 Implementing Term Rewriting

Term rewriting systems are mostly concerned with computing reduced forms of a ground term *wrt.* a set of rules. To this end, the TOM language allows the definition of many-sorted signatures that are used to generate correctly typed

algebraic terms. Further on, a rewrite system based on syntactic or equational pattern matching can be defined and applied on these terms.

First order terms. Similarly to other rule based or functional languages, TOM provides a construct to define many-sorted algebraic signatures:

```
module Peano
  Nat = zero() | suc( Nat )
```

In this example, `Peano` is the name of the module, `Nat` is a sort, `zero` and `suc` are constructors. More generally, a module may import another module, such as `Peano`, or any other predefined one like `String` or `int`. Given a signature, a well formed and well typed term can be built using the backquote (`'`) construct: `'zero()`, `'suc(zero())` are correct, as opposed to `'suc(zero(),zero())` or `'suc(3)` which are respectively not well-formed and not well-typed.

Due to the fact that TOM is built on top of JAVA, a term can be used in any JAVA expression such as `System.out.print("t = " + 'suc(suc(zero())))`. To ensure that the type of a term can be statically checked by the underlying JAVA compiler, the implementation of `'suc(suc(zero()))` should reflect the type defined by the signature. To solve this problem, we followed a different approach from other classical implementations such as ASF+SDF, OCaml, ELAN, MAUDE, ML, or Stratego.

Usually, the compiler checks that all term manipulations result in correctly typed terms, while at runtime level a generic untyped term implementation is used. In our case, we use a generator [13] that compiles the algebraic signature into a typed term structure that can be directly used by a JAVA programmer. The need for a typed term structure comes also from the fact that pure JAVA code, not handled by the TOM compiler, could create wrongly shaped terms. The generated structures are efficient, and provide types at the implementation level. As a consequence, for each sort a JAVA class with the same name is generated: `Nat t = 'suc(suc(zero()))` defines a variable `t` of sort `Nat`. By generating a statically typed implementation we provide a smooth and natural integration of the notions of signature and term into JAVA.

Pattern matching and rewriting. Implementing term rewriting may be considered a simple task. To know if a rewrite rule $l \rightarrow r$ can be applied for a ground term t , it is sufficient to have a pattern matching algorithm that computes, when it is possible, a substitution σ such that $\sigma l = t$, and fails otherwise. The application of the rule consists in replacing t by σr . However, real cases are more complicated. The rules may be applied not only on top, but also to subterms; they may have conditions; the patterns may contain symbols that belong to an equational theory (such as associativity and commutativity for example) and the application order of several rules may be prioritized. Besides, one may be interested in not only getting a single result, but also getting the set of all possible reductions of a given term t . The combinations of all these variants are difficult to tackle with. In practice, each implementation considers only a subset of them.

A main objective of TOM is to be as generic as possible. Therefore, we provide a key primitive on top of JAVA that can be used to handle most of the

situations described above: the `%match` construct. Its semantics is close to the *match* that exists in functional programming languages, but in an imperative context. A `%match` is parameterized by a list of subjects (*i.e.* expressions evaluated to ground terms) and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and fresh variables, without any restriction of linearity. The right-hand side is *not* a term, but a JAVA statement that is executed when the pattern matches the subject. But thanks to the backquote construct (```) a term can be easily built and returned. Similar to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions (*i.e.* right-hand side) may be fired for a given subject as long as no `return` or `break` is executed. To implement a simple reduction step, for each rule, we just have to encode the left-hand side by a pattern and consider a JAVA statement that returns the right-hand side. To encode a rewrite system, the notion of function that already exists in JAVA is fundamental. For example, the addition and the comparison of Peano integers may be encoded as follows:

```
public Nat plus(Nat t1, Nat t2) {
  %match(t1,t2) {
    x,zero() -> { return `x; }
    x,suc(y) -> {
      return `suc(plus(x,y));
    }
  }
}

boolean greaterThan(Nat t1, Nat t2) {
  %match(t1, t2) {
    x,x -> { return false; }
    suc(_),zero() -> { return true; }
    zero(),suc(_) -> { return false; }
    suc(x),suc(y) -> {
      return `greaterThan(x,y); }
  }}
}
```

The reader should note that anonymous variables (`_`) are allowed and that variables such as `x` or `y` do not need to be declared: they are local to each left-hand side and their type is automatically inferred.

List matching. In addition to free constructors, list operators can be also declared. They are a variant of associative operators with neutral element:

```
module Peano
  Nat      = zero() | suc( Nat )
  NatList = conc( Nat* )
```

The notation `Nat*` means that `conc` is a variadic operator where each subterm is of sort `Nat`. It can be seen as a concatenation operator over `Nat` lists: `conc()` denotes the empty list while `conc(zero(),suc(zero()))` corresponds to the list that contains `zero()` and `suc(zero())`. List operators can be used in the left-hand side of a rule in order to perform list matching:

```
Collection bag = new HashSet();
%match(list) {
  conc(*,suc(x),*) -> { bag.add(`x); }
}
System.out.println("numbers: " + bag.toString());
```

In this example, one can remark the use of *list variables*, annotated by a `*`, which intuitively corresponds to the Kleene star: such a variable is instantiated by a (possibly empty) list. Note that an action is fired for each pattern and

substitution that matches the subject. Since list matching is not unitary, the action `bag.add('x)` is evaluated for each element of `list` that matches against `suc(x)`. When applied to a list of Peano integers, this code stores each natural whose successor is in the list into the set represented by `bag`. This non-functional approach is very useful to encode non-deterministic computations such as the exploration of a search space. Combining list operators and conditions allows for the definition of complex algorithms in a concise manner, as illustrated by the following sorting algorithm:

```
public NatList sort(NatList list) {
  %match(list) {
    conc(X1*,x,X2*,y,X3*) -> {
      if(greaterThan(x,y)) { return 'sort(conc(X1*,y,X2*,x,X3*)); } }
    _ -> { return list; }
  }
}
```

Given a partially sorted list, the `sort` function looks for two elements `x` and `y` such that `x` is greater than `y`. If two such elements exist, they are swapped and the `sort` function is recursively applied. When the list is sorted this condition cannot be satisfied and the next pattern is tried: the sorted list is returned. This example also shows how a conditional rule can be naturally encoded using the `if` construct provided by JAVA.

Normal forms. When manipulating non-free algebras, it is convenient to work with terms in normal form. These normal forms are defined by a confluent and terminating rewrite system that is *systematically* applied to each term. This was the purpose of unnamed rules in ELAN for example. Instead of relying on normalization functions that have to be explicitly called by the programmer, TOM proposes to integrate the computation of normal forms into the definition of the data structure. This approach is very close to the recently introduced OCaml *private types* [11]. Normal forms are specified using the notion of *hook* which defines construction functions in the term signature. For instance, suppose that we want to work on $\mathbb{Z}/3\mathbb{Z}$. Then, we have to systematically apply the rule $suc(suc(suc(x))) \rightarrow x$ when creating new terms. This is specified by a *hook* attached to the *suc* operator.

```
module Peano
  Nat = zero() | suc( Nat )
  suc:make(t) {
    %match(t) { suc(suc(x)) -> { return 'x; } }
  }
```

Each time a *suc* is built, `make(t)` is called with `t` instantiated by the *subterm* of the considered *suc*. This is why the rewrite rule above only rewrites two *suc*. A default allocator is called when no rule can be applied

3 Controlling Rewriting

When using rewriting as a programming or modeling paradigm, it is common to consider rewrite systems that are non-confluent or non-terminating. To be able

to use them, it is necessary to exercise some control over the application of the rules. In TOM, a solution would be to use JAVA to express the control needed. While this solution provides a huge flexibility, its lack of abstraction renders difficult the reasoning about such transformations.

Rewriting based languages provide more abstract ways to express control of rule applications, by using reflexivity and the meta-level for MAUDE, or the notion of rewriting strategies for ELAN, Stratego [16], or ASF+SDF [4]. Strategies such as *bottom-up*, *top-down* or *leftmost-innermost* are higher-order features that describe how rewrite rules should be applied. We have developed a flexible and expressive strategy language inspired by ELAN, Stratego, and JJTraveler [17] where high-level strategies are defined by combining low-level primitives. For example, the *top-down* strategy is recursively defined by $\text{TopDown}(s) \triangleq \text{Sequence}(s, \text{All}(\text{TopDown}(s)))$.

Elementary strategies. An elementary strategy corresponds to a minimal transformation. It could be *Identity* (does nothing), *Fail* (always fails), or a set of *rewrite rules* (performs an elementary rewrite step only at the root position). In our system, strategies are type-preserving and have a default behavior (introduced by the keyword `extends`) that can be either `Identity` or `Fail`:

```
%strategy R() extends Fail() {
  visit Nat {
    zero()      -> { return 'suc(zero()); }
    suc(suc(x)) -> { return 'x; }
  }
}
```

When a strategy is applied to a term t , as in a `%match`, a rule is fired if a pattern matches. Otherwise, the default strategy is applied. For example, applying the strategy `R()` to the term `suc(suc(zero()))` will produce the term `zero()` thanks to the second rule. The application to `suc(suc(suc(zero())))` fails since no pattern matches at root position.

Recursive and parameterized strategies. More control is obtained by combining elementary strategies with *basic combinators* such as `Sequence`, `Choice`, `All`, `One` as presented in [2,16]. By denoting $s[t]$ the application of the strategy s to the term t , the *basic combinators* are defined as follows:

$$\begin{aligned} \text{Sequence}(s_1, s_2)[t] &\rightarrow s_2[t'] \text{ if } s_1[t] \rightarrow t' \\ &\text{failure if } s_1[t] \text{ fails} \\ \text{Choice}(s_1, s_2)[t] &\rightarrow t' \text{ if } s_1[t] \rightarrow t' \\ &s_2[t'] \text{ if } s_1[t] \text{ fails} \\ \text{All}(s)[f(t_1, \dots, t_n)] &\rightarrow f(t_1', \dots, t_n') \text{ if } s[t_1] \rightarrow t_1', \dots, s[t_n] \rightarrow t_n' \\ &\text{failure if there exists } i \text{ such that } s[t_i] \text{ fails} \\ \text{One}(s)[f(t_1, \dots, t_n)] &\rightarrow f(t_1, \dots, t_i', \dots, t_n) \text{ if } s[t_i] \rightarrow t_i' \\ &\text{failure if for all } i, s[t_i] \text{ fails} \end{aligned}$$

An example of composed strategy is $\text{Try}(s) \triangleq \text{Choice}(s, \text{Identity}())$, which applies s if it can, and performs the *Identity* otherwise. To define strategies such as *repeat*, *bottom-up*, *top-down*, etc. recursive definitions are needed.

For example, to repeat the application of a strategy s until it fails, we consider the strategy $\text{Repeat}(s) \triangleq \text{Choice}(\text{Sequence}(s, \text{Repeat}(s)), \text{Identity}())$. In TOM, we use the recursion operator μ (comparable to `rec` in OCaml) to have stand-alone definitions: $\mu x. \text{Choice}(\text{Sequence}(s, x), \text{Identity}())$.

The `All` and `One` combinators are used to define tree traversals. For example, we have $\text{TopDown}(s) \triangleq \mu x. \text{Sequence}(s, \text{All}(x))$: the strategy s is first applied on top of the considered term, then the strategy $\text{TopDown}(s)$ is recursively called on all immediate subterms of the term.

Exploration strategies. Strategy expressions can have any kind of parameters. It is common to have a `JAVA Collection` as parameter to collect some information in a tree. For example, let us consider the following strategy which collects the direct subterms of an f . This program creates a hash-set, and a strategy applied to $f(f(a()))$ collects all the subterms which are under an f : *i.e.* $\{a(), f(a())\}$.

```
%strategy Collect(c:Collection) extends Identity() {
  visit T {
    f(x) -> { c.add('x'); }
  }
}
Collection bag = new HashSet();
'TopDown(Collect(bag)).apply( 'f(f(a())) );
```

4 Meta-programming

The strategy language presented in Section 3 is very expressive and powerful to control how a set of rules should be applied. This is very convenient to collect information or traverse a tree for example. But, there is no real support to perform non-deterministic computations as in the exploration of a search space, which is essential when implementing a model checker or a prover for instance. For this purpose, we have added two new extensions to the strategy language.

Reifying $t|_{\omega}$. Given a term t , suppose that we want to compute the set of all its possible successors *wrt.* a rewrite rule $l \rightarrow r$. We have to find all possible redexes, and for each of them to compute all the substitutions that solve the matching problem. In other words, we want to compute $\{t[\sigma_1 r]_{\omega_1}, t[\sigma_2 r]_{\omega_1}, \dots, t[\sigma_p r]_{\omega_n}, \dots, t[\sigma_q r]_{\omega_n}\}$, where ω_i denotes a redex position, and σ_j is a substitution such that $\sigma_j l = t|_{\omega_i}$.

Solving this problem involves manipulating the notion of *position* in a term, and some associated operations: getting the subterm at position ω , and replacing this subterm. To the best of our knowledge, there is no rewriting based language where positions, which are internal to the implementation, can be explicitly manipulated. We introduce a new operation, `getPosition()`, which raises the notion of position at the object level, providing this global information to the level of strategies. To compute the set of all successors of t for example 4, we

¹ Many other operations and strategies may be defined.

consider the *top-down* application of a strategy parameterized by the term t itself and a collection `bag`. For each redex position ω (*i.e.* when a pattern matches), we store $t[\sigma]_{\omega}$ in `bag`, using `getPosition()` to obtain ω and `replace` to perform the replacement:

```
%strategy Collect(t:T, bag:Collection) extends Identity() {
  visit T {
    a() -> { bag.add(replace(t, getPosition(), 'b())); }
    f(x) -> { bag.add(replace(t, getPosition(), 'x')); }
    g(x) -> { bag.add(replace(t, getPosition(), 'c())); }
  }
}
T t = 'f(g(a()));
'TopDown(Collect(t,bag)).apply(t);
```

The resulting `bag` contains `f(g(b()))`, `g(a())`, and `f(c())`. This reification of the *position* notion to the object level is new in the domain of rewriting based languages, and it adds expressiveness while keeping programs close to their specification.

Rewriting a strategy. Strategy expressions are terms, and thus can themselves be subject to pattern matching and reduction by strategies. This permits dynamical adaptation of a strategy depending on the environment.

For example, suppose that we have two strategies, `s1` and `s2` which can commute. If computing `s2;s1` is more efficient than computing `s1;s2`, we can define a rule to reorder the sequences of `s1` and `s2`:

```
%strategy Reorder() extends Identity() {
  visit T {
    Sequence(s1(),s2()) -> { return 'Sequence(s2(),s1()); }
  }
}
```

This strategy can be further applied *top-down* to any strategy `s` with:
`Strategy optimized_s = 'TopDown(Reorder()).apply(s).`

5 Implementation

Since its first version in 2001, TOM itself has been written using TOM. The system is composed of a compiler and a library which defines the strategy language and offers support for predefined data-types such as integers, floats, strings, collections, and many other JAVA data-structures. The compiler is organized, in a pure functional style, as a pipeline of program transformations (type inference, simplification, compilation, optimization, generation). Each phase transforms a JAVA+TOM abstract syntax tree (AST) using rewrite rules and strategies. At the end a pure JAVA AST is obtained. The system is composed of 1000 `%match` constructs, and 200 user strategy definitions, totalizing more than 40000 lines of code. The complete environment has been integrated into Eclipse² providing

² <http://www.eclipse.org/>

a simple and efficient user interface to develop, compile, and debug rule based applications. Each component of the TOM environment is highly modular and has been designed with flexibility and reusability in mind, without introducing any performance overhead. Due to the lack of space, we mention here only a few key ingredients.

Data representation. The generator of data-structures is integrated in TOM but can be also used independently. Given a signature, it generates a set of JAVA classes that provide static typing. A subtle hash-consing technique is used to offer maximal sharing [14]: there cannot be two identical terms in memory. Therefore, the equality tests are performed in constant time, which is very efficient in many cases including when non left-linear rewrite rules are considered. In addition to the generation of lightweight data-structures, specialized hash-functions are generated for each constructor of the signature, making the generated implementation often more efficient than a hand-coded term data-structure.

Pattern matching. Thanks to the *formal anchor* approach, TOM is not restricted to a fixed term data-structure. We have designed a compilation algorithm where the data-structure is a parameter of the pattern matching algorithm. In particular, TOM can be used to match and rewrite XML documents. Moreover, the underlying host language is also a parameter, making possible the compilation (including list-matching) into different target languages such as C, JAVA, OCaml, and Python. This approach has been formally studied in [10]. Besides, for each compilation of a set of patterns, TOM provides a COQ proof that the generated code is correct *wrt.* the semantics of pattern matching.

Strategies. Most of the strategy library is written in TOM, making its extension easy. Only a few of low-level elementary strategies (Sequence, Choice, All, One, *etc.*) are implemented in JAVA. However, the considered object design-patterns for these elementary strategies facilitate their extension also. To add a new probabilistic choice operator for example, less than 30 lines of code have to be written, without any re-compilation of the system. This makes TOM an ideal platform to experiment new paradigms. Once again, the methodology used to implement the strategy library is not restricted to a given term representation, being possible to switch to another one by properly applying the *interface* concept offered by JAVA.

6 Applications

The TOM system is no longer a prototype. It has been used to implement many large and complex applications, among them the compiler itself. It has also been used in an industrial context to implement a query optimizer for Xquery, a platform for transforming and analysing timed automata using XML manipulation, *etc.* In this section we focus on some applications where the key characteristics of TOM were particularly useful.

Proof assistant. *lemuridae* is a proof assistant for superdeduction [6] (a dynamic extension of sequent calculus). Proof trees benefit from maximal memory sharing

which allows for the handling of big proofs while tactics are naturally translated into strategies. Besides, the expressiveness of TOM patterns makes the micro-proofchecker only one hundred lines long and therefore enables a high degree of confidence in the prover.

NSPK. TOM has been used to implement the verification of the Needham-Schroeder public key protocol by model checking. Several strategies have been experimented. The resulting implementation can be compared favorably with state of the art rule based implementation such as MAUDE or Mur φ .

Rho-calculus. An interpreter for the rho-calculus [7] with explicit substitutions was developed using TOM. It is surprisingly short and close to the operational semantics of the calculus taking advantage of all the capabilities of TOM. The calculus itself is expressed using rewrite rules and parameterized strategies, while the interactive features and user interface operations take advantage of the underlying JAVA language.

Calculus of structures. TOM is used to implement an automatic prover for the system BV in the calculus of structures [13]. Normal forms are used to implement the several associative-commutative operators with neutral elements while first-order positions allow to manage the high level of non determinism introduced by deep inference.

On several classical benchmarks TOM is competitive with state of the art implementations like ASF+SDF, ELAN, or MAUDE³. In the following, *Fibonacci* computes 500 times the 18th Fibonacci number using a Peano representation. *Sieve* computes prime numbers up to 2000 using list matching to eliminate non-prime numbers: $(c_1*, x, c_2*, y, c_3*) \rightarrow (c_1*, x, c_2*, c_3*)$ if x divides y ⁴. *Evalsym*, *Evalexp*, and *Evaltree* are three benchmarks based on the normalization of the expression $2^{22} \bmod 17$ using different reduction strategies. These three benchmarks were first presented in [5]. All these examples are available on the TOM source repository⁵. The measurements were done on a MacBook Pro 2.33 GHz, using Java 1.5.0 and gcc 4.0.

	Fibonacci	Sieve	Evalsym	Evalexp	Evaltree
ASF + SDF	0.4 s	24.1 s	1.7 s	2.0 s	1.6 s
ELAN	1.1 s	–	5.3 s	11.8 s	10.1 s
MAUDE	2.3 s	17.7 s	8.8 s	15.4 s	21.3 s
TOM C	0.6 s	0.2 s	1.9 s	2.0 s	2.2 s
TOM Java	1.9 s	2.2 s	7.8 s	8.4 s	8.2 s

³ Note that MAUDE is an *interpreter*. The experimental results are extraordinarily good compared to the compiled and highly optimized low-level C implementations.

⁴ On this example, the performance of ASF+SDF may be explained by the lack of support for builtin integers.

⁵ <http://gforge.inria.fr/projects/tom/>

7 Related Work

The creation of TOM was motivated by the difficulty in integrating or reusing a term rewriting based language in an industrial context. The initial idea of piggybacking rewriting on a generalist host language was inspired by Lex and Yacc and generalized in the framework of formal islands. TOM is the result of a long term effort, on one hand integrating innovative ideas and concepts, and on the other hand combining and incorporating key notions and techniques developed in well reputed research teams. Regarding the data-structures, the implementation of the generator has been done in strong cooperation with the authors of ApiGen [15] and followed our experience with the ATerm [14] library used by ASF+SDF. The originality of our solution is to provide typed constructs resulting in a faster and safer implementation. Moreover, we introduce the notion of *hook* which is strongly related to OCaml *private types*. Concerning matching theories, TOM is similar to ASF+SDF by providing list-matching, that corresponds to associative matching with neutral element. Contrary to MAUDE and ELAN, the restriction to list-matching instead of more complex theories like associative-commutative makes the implementation simpler and powerful enough in many cases. The design of the strategy language has been inspired by ELAN, Stratego, and JTraveler. Compared to ELAN, TOM does not support implicit non-deterministic strategies, implemented using back-tracking. But due to reification of $t_{|\omega}$, explicit non-deterministic computations are practical. ASF+SDF does not have a strategy language but provides traversal functions that can be used to control how a set of rule should be applied. By raising the notion of strategy to the object level, TOM offers meta-programming capabilities that may remind the meta-level of MAUDE. With regard to strategy languages, Stratego is certainly the language to which TOM is the most close, the main differences being strategies as terms and explicit non-deterministic computations.

8 Conclusion

In this paper we introduced the system TOM, which brings rewriting techniques to the world of mainstream programming languages. In addition to this original result, the contributions of TOM include: the notion of *formal island*; the certification of pattern matching; a support for *private types* in JAVA; an efficient implementation of typed and maximally shared terms; user definable recursive strategies (in JAVA) using the μ operator; strategies considered as terms; reification of $t_{|\omega}$ that makes non-deterministic computations explicit.

We are currently working on two extensions. One is the formalization and the integration of the notion of anti-patterns [9], which enables the expression of negative constraints inside patterns. The second one concerns the integration of termgraph rewriting capabilities into the language and the extension of the strategy library.

References

1. Balland, E., Kirchner, C., Moreau, P.-E.: Formal islands. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 51–65. Springer, Heidelberg (2006)
2. Borovanský, P., Kirchner, C., Kirchner, H.: Controlling rewriting by rewriting. In: Meseguer, J. (ed.) Proceedings of WRLA 1996. ENTCS, vol. 4, Elsevier Science Publishers, North-Holland, Amsterdam (1996)
3. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., Ringeissen, C.: An overview of ELAN. In: Kirchner, C., Kirchner, H. (eds.) Proceedings of WRLA 1998. ENTCS, vol. 15, Elsevier Science Publishers, Amsterdam (1998)
4. Brand, M., Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Wilhelm, R. (ed.) CC 2001 and ETAPS 2001. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
5. Brand, M., Klint, P., Olivier, P.: Compilation and Memory Management for ASF+SDF. *Compiler Construction* 1575, 198–213 (1999)
6. Brauner, P., Houtmann, C., Kirchner, C.: Principles of superdeduction. In: LICS, (To appear) (January 2007)
7. Cirstea, H., Kirchner, C.: The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure. and Applied Logics* 9(3), 427–498 (2001)
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
9. Kirchner, C., Kopetz, R., Moreau, P.: Anti-pattern matching. In: Proceedings of the 16th European Symposium on Programming. LNCS, vol. 4421, pp. 110–124. Springer, Heidelberg (2007)
10. Kirchner, C., Moreau, P.-E., Reilles, A.: Formal validation of pattern matching code. In: Barahone, P., Felty, A. (eds.) Proceedings of PPDP 2005, pp. 187–197. ACM Press, New York (2005)
11. Leroy, X.: The objective caml system release 3.09 <http://caml.inria.fr/>
12. Moreau, P.-E., Ringeissen, C., Vittek, M.: A Pattern Matching Compiler for Multiple Target Languages. In: Hedin, G. (ed.) CC 2003 and ETAPS 2003. LNCS, vol. 2622, pp. 61–76. Springer, Heidelberg (2003)
13. Reilles, A.: Canonical abstract syntax trees. In: Proceedings of WRLA 2006 ENTCS (To appear) (2006)
14. van den Brand, M., de Jong, H., Klint, P., Olivier, P.: Efficient annotated terms. *Software, Practice and Experience* 30(3), 259–291 (2000)
15. van den Brand, M., Moreau, P.-E., Vinju, J.: A generator of efficient strongly typed abstract syntax trees in Java. In: IEE Proceedings - Software Engineering, vol. 152(2) pp. 70–78 (December 2005)
16. Visser, E., Benaïssa, Z.-e.-A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 13–26. ACM Press, New York, USA (1998)
17. Visser, J.: Visitor combination and traversal control. In: Proceedings of the 16th ACM SIGPLAN conference on OOPSLA, pp. 270–282. ACM Press, NY, USA (2001)

Rewriting Approximations for Fast Prototyping of Static Analyzers

Yohan Boichut, Thomas Genet, Thomas Jensen, and Luka Le Roux*

Université de Rennes 1 and INRIA and CNRS
IRISA, Campus de Beaulieu, F-35042 Rennes, France
{boichut,genet,jensen,leroux}@irisa.fr

Abstract. This paper shows how to construct static analyzers using tree automata and rewriting techniques. Starting from a term rewriting system representing the operational semantics of the target programming language and given a program to analyze, we automatically construct an over-approximation of the set of reachable terms, i.e. of the program states that can be reached. The approach enables fast prototyping of static analyzers because modifying the analysis simply amounts to changing the set of rewrite rules defining the approximation. A salient feature of this approach is that the approximation is correct by construction and hence does not require an explicit correctness proof. To illustrate the framework proposed here on a realistic programming language we instantiate it with the Java Virtual Machine semantics and perform class analysis on Java bytecode programs.

1 Introduction

The aim of this paper is to show how to combine rewriting theory with principles from abstract interpretation in order to obtain a fast and reliable methodology for implementing static analyzers for programs. Rewriting theory and in particular reachability analysis based on tree automata has proved to be a powerful technique for analyzing particular classes of software such as cryptographic protocols [11,8,12]. In this paper we set up a framework that allows to apply those techniques to a general programming language. Our framework consists of three parts. First, we define an encoding of the operational semantics of the language as a term rewriting system (TRS for short). Second, we give a translation scheme for transforming programs into rewrite rules. Finally, an over-approximation of the set of reachable program states represented by a tree automaton is computed using the tree automata completion algorithm [8]. In this paper, we instantiate this framework on a real test case, namely Java. We encode the Java Virtual Machine (JVM for short) operational semantics and Java bytecode programs into TRS and construct over-approximations of JVM states.

With regards to rewriting, the contribution of this paper is dual. First, we propose an encoding of a significant part of Java into *left-linear, unconditional*

* Funded by France Telecom CRE 46128352.

TRS. For rewriting, the second contribution is to have scaled up the theoretical construction of tree automata completion to the verification of Java bytecode programs. With respect to static analysis, the contribution of this paper is to show that regular approximations can be used as a foundational mechanism for ensuring, by construction, safety of static analyzers. This paper shows that the approach can already be used to achieve standard class analysis on Java bytecode programs. Moreover, using approximation rules instead of abstract domains makes the analysis easier to fine-tune and to prove correct. This is of great interest, when a standard analysis is too coarse, since our technique permits to adapt the analysis to the property to prove and preserve safety.

The paper is organized as follows. Section 2 introduces the formal background of the rewriting theory. Section 3 shows how to over-approximate the set of reachable terms using tree automata. Section 4 presents a term rewriting model of the Java semantics. Section 5 presents, by the means of some classical examples, how rewriting approximations can be used for a class analysis. Section 6 compares our contribution with related works. Section 7 concludes.

2 Formal Background

Comprehensive surveys can be found in [6,11] for term rewriting systems, and in [5,14] for tree automata and tree language theory.

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 $\text{Var}(t)$. 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 $\text{Pos}(t)$ of positions of a term t is inductively defined by:

- $\text{Pos}(t) = \{\epsilon\}$ if $t \in \mathcal{X}$
- $\text{Pos}(f(t_1, \dots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \text{Pos}(t_i)\}$

If $p \in \text{Pos}(t)$, 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 \mathcal{R} is a set of *rewrite rules* $l \rightarrow r$, where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\text{Var}(l) \supseteq \text{Var}(r)$. A rewrite rule $l \rightarrow r$ 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 $l \rightarrow r$ of \mathcal{R} is left-linear). The TRS \mathcal{R} induces a rewriting relation $\rightarrow_{\mathcal{R}}$ on terms whose reflexive transitive closure is denoted by $\rightarrow_{\mathcal{R}}^*$. The set of \mathcal{R} -descendants of a set of ground terms E is $\mathcal{R}^*(E) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in E \text{ s.t. } s \rightarrow_{\mathcal{R}}^* t\}$.

The verification technique we propose in this paper is based on the computation of $\mathcal{R}^*(E)$. 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 [14]. However, it is possible to over-approximate it [8,18] using tree automata, i.e. a finite representation of infinite (regular) sets of terms. We next 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 $c \rightarrow q$, 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 $c \rightarrow q$ where $c = f(q_1, \dots, q_n)$, $f \in \mathcal{F}$ whose arity is n , and $q_1, \dots, q_n \in \mathcal{Q}$.

Definition 2 (Bottom-up non-deterministic finite tree automaton). A bottom-up non-deterministic 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.

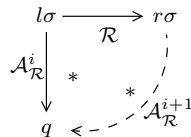
The rewriting relation on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transitions of \mathcal{A} (the set Δ) is denoted by \rightarrow_{Δ} . When Δ is clear from the context, \rightarrow_{Δ} will also be denoted by $\rightarrow_{\mathcal{A}}$.

Definition 3 (Recognized language). The tree language recognized by \mathcal{A} in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \xrightarrow{*}_{\mathcal{A}} q\}$. 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 Approximations of Reachable Terms

Given a tree automaton \mathcal{A} and a TRS \mathcal{R} , the tree automata completion algorithm, proposed in [10, 8], computes a tree automata completion $\mathcal{A}_{\mathcal{R}}^k$ such that $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^k) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for the classes of TRSs where an exact computation is possible, see [8]) and such that $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

The tree automata completion works as follows. From $\mathcal{A} = \mathcal{A}_{\mathcal{R}}^0$ completion builds a sequence $\mathcal{A}_{\mathcal{R}}^0, \mathcal{A}_{\mathcal{R}}^1, \dots, \mathcal{A}_{\mathcal{R}}^k$ of automata such that if $s \in \mathcal{L}(\mathcal{A}_{\mathcal{R}}^i)$ and $s \rightarrow_{\mathcal{R}} t$ then $t \in \mathcal{L}(\mathcal{A}_{\mathcal{R}}^{i+1})$. If we find a fixpoint automaton $\mathcal{A}_{\mathcal{R}}^k$ such that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_{\mathcal{R}}^k)) = \mathcal{L}(\mathcal{A}_{\mathcal{R}}^k)$, then we have $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^k) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}_{\mathcal{R}}^0))$ (or $\mathcal{L}(\mathcal{A}_{\mathcal{R}}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$) if \mathcal{R} is not in one class of [8]. To build $\mathcal{A}_{\mathcal{R}}^{i+1}$ from $\mathcal{A}_{\mathcal{R}}^i$, we achieve a *completion step* which consists of finding *critical pairs* between $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{A}_{\mathcal{R}}^i}$. 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 $l\sigma \xrightarrow{*}_{\mathcal{A}_{\mathcal{R}}^i} q$ and $l\sigma \rightarrow_{\mathcal{R}} r\sigma$. Note that since \mathcal{R} , $\mathcal{A}_{\mathcal{R}}^i$ and the set \mathcal{Q} of states of $\mathcal{A}_{\mathcal{R}}^i$ are finite, there is only a finite number of critical pairs. For every critical pair detected between \mathcal{R} and $\mathcal{A}_{\mathcal{R}}^i$ such that $r\sigma \not\xrightarrow{*}_{\mathcal{A}_{\mathcal{R}}^i} q$, the tree automaton $\mathcal{A}_{\mathcal{R}}^{i+1}$ is constructed by adding a new transition $r\sigma \rightarrow q$ to $\mathcal{A}_{\mathcal{R}}^i$ such that $\mathcal{A}_{\mathcal{R}}^{i+1}$ recognizes $r\sigma$ in q , i.e. $r\sigma \rightarrow_{\mathcal{A}_{\mathcal{R}}^{i+1}} q$.



However, the transition $r\sigma \rightarrow q$ is not necessarily a normalized transition of the form $f(q_1, \dots, q_n) \rightarrow q$ and so it has to be normalized first. Since normalization consists in associating state symbols to subterms of the left-hand side of the new transition, it always succeeds. Note that, when using new states to normalize the transitions, completion is as precise as possible. However, without approximation, completion is likely not to terminate (because of general undecidability results [14]). To enforce termination, and produce an over-approximation, the completion algorithm is parametrized by a set N of *approximation rules*. When the set N is used during completion to normalize transitions, the obtained tree automata are denoted by $\mathcal{A}_{N,\mathcal{R}}^1, \dots, \mathcal{A}_{N,\mathcal{R}}^k$. Each such rule describes a context in which a list of rules can be used to normalize a term. For all $s, l_1, \dots, l_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and for all $x, x_1, \dots, x_n \in \mathcal{Q} \cup \mathcal{X}$, the general form for an approximation rule is:

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow x_1, \dots, l_n \rightarrow x_n].$$

The expression $[s \rightarrow x]$ is a pattern to be matched with the new transitions $t \rightarrow q'$ obtained by completion. The expression $[l_1 \rightarrow x_1, \dots, l_n \rightarrow x_n]$ is a set of rules used to normalize t . To normalize a transition of the form $t \rightarrow q'$, we match s with t and x with q' , obtain a substitution σ from the matching and then we normalize t with the rewrite system $\{l_1\sigma \rightarrow x_1\sigma, \dots, l_n\sigma \rightarrow x_n\sigma\}$. Furthermore, if $\forall i = 1 \dots n : x_i \in \mathcal{Q}$ or $x_i \in \text{Var}(l_i) \cup \text{Var}(s) \cup \{x\}$ then since $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, $x_1\sigma, \dots, x_n\sigma$ are necessarily states. If a transition cannot be fully normalized using approximation rules N , normalization is finished using some new states, see Example 1.

The main property of the tree automata completion algorithm is that, whatever the state labels used to normalize the new transitions, if completion terminates then it produces an over-approximation of reachable terms [8].

Theorem 1 ([8]). *Let \mathcal{R} be a left-linear TRS, \mathcal{A} be a tree automaton and N be a set of approximation rules. If completion terminates on $\mathcal{A}_{N,\mathcal{R}}^k$ then*

$$\mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

Here is a simple example illustrating completion and the use of approximation rules when the language $\mathcal{R}^*(E)$ is not regular.

Example 1. Let $\mathcal{R} = \{g(x, y) \rightarrow g(f(x), f(y))\}$ and let \mathcal{A} be the tree automaton such that $\mathcal{Q}_f = \{q_f\}$ and $\Delta = \{a \rightarrow q_a, g(q_a, q_a) \rightarrow q_f\}$. Hence $\mathcal{L}(\mathcal{A}) = \{g(a, a)\}$ and $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{g(f^n(a), f^n(a)) \mid n \geq 0\}$. Let $N = [g(f(x), f(y)) \rightarrow z] \rightarrow [f(x) \rightarrow q_1, f(y) \rightarrow q_1]$. During the first completion step, we find a critical pair $g(q_a, q_a) \rightarrow_{\mathcal{R}} g(f(q_a), f(q_a))$ and $g(q_a, q_a) \rightarrow_{\mathcal{A}}^* q_f$. We thus have to add the transition $g(f(q_a), f(q_a)) \rightarrow q_f$ to \mathcal{A} . To normalize this transition, we match $g(f(x), f(y))$ with $g(f(q_a), f(q_a))$ and match z with q_f and obtain $\sigma = \{x \mapsto q_a, y \mapsto q_a, z \mapsto q_f\}$. Applying σ to $[f(x) \rightarrow q_1, f(y) \rightarrow q_1]$ gives $[f(q_a) \rightarrow q_1, f(q_a) \rightarrow q_1]$. This last system is used to normalize the transition $g(f(q_a), f(q_a)) \rightarrow q_f$ into the set $\{g(q_1, q_1) \rightarrow q_f, f(q_a) \rightarrow q_1\}$ which is added

to \mathcal{A} to obtain $\mathcal{A}_{N,\mathcal{R}}^1$. The completion process continues for another step and ends on $\mathcal{A}_{N,\mathcal{R}}^2$ whose set of transition is $\{a \rightarrow q_a, g(q_a, q_a) \rightarrow q_f, g(q_1, q_1) \rightarrow q_f, f(q_a) \rightarrow q_1, f(q_1) \rightarrow q_1\}$. We have $\mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k) = \{g(f^n(a), f^m(a)) \mid n, m \geq 0\}$ which is an over-approximation of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

The tree automata completion algorithm and the approximation mechanism are implemented in the Timbuk [13] tool. On the previous example, once the fixpoint automaton $\mathcal{A}_{N,\mathcal{R}}^k$ has been computed, it is possible to check whether some terms are reachable, i.e. recognized by $\mathcal{A}_{N,\mathcal{R}}^k$ or not. This can be done using tree automata intersections [8]. Another way to do that is to search instances for a pattern t , where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, in the tree automaton. Given t it is possible to check if there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a state $q \in \mathcal{Q}$ such that $t\sigma \xrightarrow{*}_{\mathcal{A}_{N,\mathcal{R}}^k} q$. If such a solution exists then it proves that there exists a term $s \in \mathcal{T}(\mathcal{F})$, a position $p \in \text{Pos}(s)$ and a substitution $\sigma' : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $s[t\sigma']_p \in \mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. that $t\sigma'$ occurs as a subterm in the language recognized by $\mathcal{L}(\mathcal{A}_{N,\mathcal{R}}^k)$. On the other hand, if there is no solution then it proves that no such term is in the over-approximation, hence it is not in $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, i.e. it is not reachable.

In the patterns we use in this paper, $?x$ denotes variables for which a value is wanted and $'_$ denotes anonymous variables for which no value is needed. For instance, the pattern $g(f(-), g(-, -))$ has no solution on $\mathcal{A}_{N,\mathcal{R}}^2$ of example 11, meaning that no term containing any ground instance of this pattern is reachable by rewriting $g(a, a)$.

4 Formalization of the Java Bytecode Semantics Using Rewriting Rules

This section describes how to formalize the semantics of an object-oriented language (here, Java bytecode) using rewriting rules. From a bytecode Java program p , we have developed a prototype that automatically produces a TRS \mathcal{R} modeling a significant part of the Java semantics (stacks, frames, objects, references, methods, heaps, integers) as well as the semantics of p . For the moment, exceptions and threads are not taken into account but they can be elegantly encoded using rewriting [16,7]. The formalization follows the structure of standard Java semantics formalizations [2,9].

4.1 Formalization of Java Program States

A Java program state contains a current execution frame (also called activation record), a frame stack, a heap, and a static heap. A frame gives information about the method currently being executed: its name, current program counter, operand stack and local variables. When a method is invoked the current frame is stored in the frame stack and a new current frame is created. A heap is used to store objects and arrays, i.e. all the information that is not local to the execution

of a method. The static heap stores values of static fields, i.e. values that are shared by all objects of a same class.

Let P be the infinite set of all the possible Java programs. Given $p \in P$, let $C(p)$ be the corresponding finite set of class identifiers and $C_r(p)$ be $C(p) \cup \{\text{array}\}$. A value is either a primitive type or a reference pointing to an object (or an array) in the heap. In our setting, we only consider integer and boolean primitive types. Let $PC(p)$ be the set of integers from 0 to the highest possible program point in all the methods in p . Let $M(p)$ be the set of method names and $M_{id}(p)$ be the finite set of pairs (m, c) where $m \in M(p)$, $c \in C(p)$ and m is a method defined by the class c . This last set is needed to distinguish between methods having the same name but defined by different classes. For the sake of simplicity, we do not distinguish between methods having the same name but a different signature but this could easily be done.

Following standard Java semantics we define a *frame* to be a tuple $f = (pc, m, s, l)$ where $pc \in PC(p)$, $m \in M_{id}(p)$, s an operand stack, l a finite map from indexes to values (local variables). An object from a class c is a map from field identifiers to values. The heap is a map from references to objects and arrays. The static heap is a map from static field names to values. A program state is a tuple $s = (f, fs, h, k)$ where f is a frame, fs is a stack of frames, h is a heap and k a static heap.

4.2 A Program State as a Term

Let $\mathcal{F}_C(p) = C(p)$ and $\mathcal{F}_{C_r}(p) = C_r(p) = C(p) \cup \{\text{array}\}$ be sets of symbols. We encode a reference as a term $loc(c, a)$ where $c \in C_r(P)$ is the class of the object being referenced and a is an integer. This is coherent with Java semantics where it is always possible to know dynamically the class of an object corresponding to a reference. We use $\mathcal{F}_{primitive} = \{\text{succ} : 1, \text{pred} : 1, \text{zero} : 0\}$ for primitive types (integers), $\mathcal{F}_{reference}(p) = \{loc : 2, \text{succ} : 1, \text{zero} : 0\} \cup \mathcal{F}_{C_r}(p)$ for references and $\mathcal{F}_{value}(p) = \mathcal{F}_{primitive} \cup \mathcal{F}_{reference}(p)$ for values. For example, $loc(\text{foo}, \text{succ}(\text{zero}))$ is a reference pointing to the object located at the index 1 in the *foo* class heap. Let x be the higher program point of the program (p), then $\mathcal{F}_{PC}(p) = \{pp0 : 0, pp1 : 0, \dots, pp_x : 0\}$. $\mathcal{F}_M(p)$ is defined the same way as $\mathcal{F}_C(p)$. $\mathcal{F}_{M_{id}}(p) = \{\text{name} : 2\} \cup \mathcal{F}_M(p) \cup \mathcal{F}_C(p)$. For example $\text{name}(\text{bar}, A)$ stands for the method *bar* defined by the class *A*. Let $l(p)$ denote the maximum of local variables used by the methods of the program package p . We use $\mathcal{F}_{stack}(p) = \{\text{stack} : 2, \text{nilstack} : 0\} \cup \mathcal{F}_{value}(p)$ for stacks, $\mathcal{F}_{localVars}(p) = \{\text{locals} : l(p), \text{nillocal} : 0\} \cup \mathcal{F}_{value}(p)$ for local variables and $\mathcal{F}_{frame}(p) = \{\text{frame} : 4\} \cup \mathcal{F}_{PC}(p) \cup \mathcal{F}_{M_{id}}(p) \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{localVars}(p)$ for frames. A possible frame thus would be: $\text{frame}(\text{name}(\text{bar}, A), pp4, \text{stack}(\text{succ}(\text{zero}), \text{nilstack}), \text{locals}(\text{loc}(\text{bar}, \text{zero}), \text{nillocal}))$ where the program counter points to the 4th instruction of the method *bar* defined by the class *A*. The current operand stack has the integer 1 on the top. The first local variable is some reference and the other is not initialized.

The alphabet $\mathcal{F}_{objects}(p)$ contains the same symbols as $\mathcal{F}_C(p)$, where the arity of each symbol is the corresponding number of non-static fields. As an example, $\text{object}A(\text{zero})$ is an object from the class *A* with one field whose value is *zero*.

Let nc be the number of classes. We chose to divide the heap into nc *class heaps* plus one for the arrays. In a reference $loc(c, a)$, a is the index of the object in the list representing the heap of class c . An array is encoded using a list and indexes in a similar way. We use $\mathcal{F}_{heap}(p) = \{heaps : (nc + 1), heap : 2\} \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{objects}(p)$ for heaps, and $\mathcal{F}_{state}(p) = \{state : 4\} \cup \mathcal{F}_{frame}(p) \cup \mathcal{F}_{heap}(p)$ for states.

4.3 Java Bytecode Semantics

Figure 1 presents some rules of the semantics operating at the frame level. For a given instruction, if a frame matches the top expression then it is transformed into the lower expression. Considering the frame (pc, m, s, l) , pc denotes the current program point, m the current method identifier, s the current stack and l the current array of local variables. The operator $::$ models stack concatenation. The $store_i$ instruction is used to store the value at the top of the current stack in the i^{th} register, where $x \rightarrow_i l$ denotes the new resulting array of local variables.

$$\boxed{\begin{array}{c} (pop) \frac{(m, pc, x :: s, l)}{(m, pc + 1, s, l)} \quad (store_i) \frac{(m, pc, x :: s, l)}{(m, pc + 1, s, x \rightarrow_i l)} \end{array}}$$

Fig. 1. Example of bytecodes operating at the frame level

Figure 2 presents a rule of the semantics operating at the state level. For a state $((m, pc, s, l), fs, h, k)$, the symbols pc , m , s and l denote the current frame components, fs the current stack of frames, h the heap and k the static heap. The instruction $invokeVirtual_{name}$ implements dynamic method invocation. The method to be invoked is determined from its $name$ and the class of the reference at the top of the stack. The internal function $class(ref, h, k)$ is used to get the reference's class c and $lookup(name, c)$ searches the class hierarchy in a bottom-up fashion for the the method m' corresponding to this name and this class. There are internal functions to manage the parameters of the method (pushed on the stack before invoking): $storeparams(ref :: s, m')$ to build an array of local variables from values on the top of the operand stack and $popparams(ref :: s, m')$ to remove from the current operand stack the parameters used by m' . With those tools, it is possible to build a new frame pointing at the first program point of m' and to push the current frame on the frame stack. Some other examples can be found in [3].

$$\boxed{\begin{array}{c} (invokeVirtual_{name}) \\ \frac{((m, pc, ref :: s, l), fs, h, k), c = class(ref, h, k), m' = lookup(name, c)}{((m', 0, [], storeparams(ref :: s, m')), (m, pc + 1, popparams(ref :: s, m'), l) :: fs, h, k)} \end{array}}$$

Fig. 2. Example of bytecodes operating at the state level

4.4 Java Bytecode Semantics Using Rewriting Rules

In this section, we encode the operational semantics into rewriting rules in a way that makes the resulting system amenable to approximation by the techniques described in this paper. The first constraint is that the term rewriting system has to be left-linear (see Theorem [1](#)). The second constraint, is that intermediate steps modeling internal operations of the JVM (such as low level rewriting for evaluating arithmetic operations $+$, $*$, \dots), should be easy to filter out. To this end, we introduce a notion of intermediate frames (named *xframe*) encompassing all the internal computations performed by the JVM, which are not part of operational semantic rules. We can express the Java Bytecode Semantics of a Java bytecode program p by means of rewriting rules (see [3](#) details). We give here the encodings of *pop* and *invokeVirtual* instructions.

In the following, symbols $m, c, pc, s, l, fs, h, k, x, y, a, b, adr, l0, l1, l2, size, h, h0, h1, ha$ are variables. For a given program point pc in a given method m , we build an *xframe* term very similar to the original *frame* term but with the current instruction explicitly stated. The *xframes* are used to compute intermediate steps. If an instruction requires several internal rewriting steps, we will only rewrite the corresponding *xframe* term until the execution of the instruction ends. Assume that, in program p , the instruction at program point $pp2$ of method *foo* of class A is *pop*. In figure [3](#), Rule 1 builds a *xframe* term by explicitly adding the current instruction to the *frame* term. Rule 2 describes the semantics of *pop*. Rule 3 specifies the control flow by defining the next program point.

1	$frame(name(foo, A), pp2, s, l) \rightarrow xframe(pop, name(foo, A), pp2, s, l)$
2	$xframe(pop, m, pc, stack(x, s), l) \rightarrow frame(m, next(pc), s, l)$
3	$next(pp2) \rightarrow pp3$

Fig. 3. An example *pop* instruction by rewriting rules, for program p

Now, assume that, in program p , the instruction at program point $pp2$ of method *foo* of class A is *invokeVirtual*. This instruction requires to compile some information about methods and the class hierarchy into the rules. Basically, we need to know what is the precise method to invoke, given a class identifier and a method name. In p , assume that A and B are two classes such that B extends A . Let *set* be a method implemented in the class A (and thus available from B) with one parameter and *reset* a method implemented in the class B (and thus unavailable from A) with no parameter. Figure [4](#) presents the resulting rules for this simple example. To complete the modeling of the semantics and the program by rewriting rules we need stubs for native libraries used by the program. At present, we have developed stubs for some of the methods from the *javaioInputStream* class. We model interactions of a Java program state with its environment using a term of the form $IO(s, i, o)$ where s is the state, i is the input stream and o the output stream.

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow xframe(invokeVirtual(set), name(foo, A), pp2, s, l)$
2	$state(xframe(invokeVirtual(set), m, pc, stack(loc(A, adr), stack(x, s)), l), fs, h, k)$	$\rightarrow state(frame(name(set, A), pp0, s, locals(loc(A, adr), x, nillocal)), stack(storedframe(m, pc, s, l), fs), h, k)$
3	$state(xframe(invokeVirtual(set), m, pc, stack(loc(B, adr), stack(x, s)), l), fs, h, k)$	$\rightarrow state(frame(name(set, A), pp0, s, locals(loc(B, adr), x, nillocal)), stack(storedframe(m, pc, s, l), fs), h, k)$
4	$state(xframe(invokeVirtual(reset), m, pc, stack(loc(B, adr), s), l), fs, h, k)$	$\rightarrow state(frame(name(reset, B), pp0, s, locals(loc(B, adr), nillocal, nillocal)), stack(storedframe(m, pc, s, l), fs), h, k)$
5	$next(pp2)$	$\rightarrow pp3$

Fig. 4. $invokeVirtual_{set}$ instruction by rewriting rules

5 Class Analysis as a Rewriting Theory

In most program analyzes, it is often necessary to know the control flow graph. For Java, as for other object-oriented languages, the control flow depends on the data flow. When a method is invoked, to know which one is executed, the class of the involved object is needed. For instance, on the Java program of Figure 5, $x.foo()$ calls $this.bar()$. To know which version of the `bar` is called, it is necessary to know the class of `this` and thus the class of `x` in $x.foo()$ call. The method actually invoked is determined dynamically during the program run. Class analysis aims at statically determining the class of objects stored in fields and local variables, and allows to build a more precise control flow graph valid for all possible executions. Note that in this example, exceptions around `System.in.read()` are required by the Java compiler. However, in this paper, we do not take them into account in the control flow.

There are different standard class analyzes, from simple and fast to precise and expensive. We consider k -CFA analysis [17]. In these analyzes, primitive types are abstracted by the name of their type and references are abstracted by the class of the objects they point to. In 0-CFA analysis, each method is analyzed only once, without distinguishing between the different calls (and hence the arguments passed) to this method. k -CFA analyzes different calls to the same method separately, taking into account up to k frames on the top of the frame stack.

Starting from a term rewriting system \mathcal{R} modeling the semantics of a Java program, and a tree automaton \mathcal{A} recognizing a set of initial Java program states, we aim at computing an automaton $\mathcal{A}_{N, \mathcal{R}}^k$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. We developed a prototype which produces \mathcal{R} and \mathcal{A} from a Java `.class` file. From the Java source program of Figure 5, one can obtain the files `Test.class`, `A.class` and `B.class` whose content is around 90 lines of bytecode. The TRS \mathcal{R} produced by compilation of those classes is composed of 275 rewrite rules. The number of rewrite rules is linear w.r.t. the size of the bytecode files. The analysis itself is performed using Timbuk [13]. Successively, this section details a

0-CFA, a 1-CFA and an even more precise analysis obtained using the same TRS \mathcal{R} and automaton \mathcal{A} , but using different sets of approximation rules. On this program, the set of reachable program states is infinite (and thus approximations are necessary) because the instruction `x=System.in.read()`, reading values in the input stream, is embedded in an unbounded loop. As long as the value stored in the variable `x` is different from 0, the computation continues. Moreover, since we want to analyze this program for any possible stream of integers, in the automaton \mathcal{A} the input stream is unbounded.

```

class A{
    int y;
    void foo(){this.bar();}
    void bar(){y=1;}
}
class B extends A{
    void bar(){y=2;}
}
class Test{
    public void execute(A x){
        x.foo();
    }
    public void main(String[] argv){
        A o1;
        B o2;
        int x;
        o1= new A();
        o2= new B();
        try{
            x=System.in.read();
        }
        catch (java.io.IOException e)
            { x = 0;}
        while (x != 0){
            execute(o1);
            execute(o2);
            try{
                x=System.in.read(); }
            catch (java.io.IOException e)
                { x = 0;}
        }
    }
}

```

Fig. 5. Java Program Example

5.1 0-CFA Analysis

For a 0-CFA analysis, all integers are abstracted by their type, i.e. they are defined by the following transitions in \mathcal{A} : $zero \rightarrow q_{int}$, $succ(q_{int}) \rightarrow q_{int}$ and $pred(q_{int}) \rightarrow q_{int}$. The input stream is also specified by \mathcal{A} as an infinite stack of integers: $nilstackin \rightarrow q_{in}$ and $stackin(q_{int}, q_{in}) \rightarrow q_{in}$. Approximation rules for integers, streams and references are defined by: $[x \rightarrow y] \rightarrow [zero \rightarrow q_{int}, succ(q_{int}) \rightarrow q_{int}, pred(q_{int}) \rightarrow q_{int}, nilstackin \rightarrow q_{in}, stackin(q_{int}, q_{in}) \rightarrow q_{in}, loc(A, \alpha) \rightarrow q_{refA}, loc(B, \beta) \rightarrow q_{refB}]$ where x, y, α and β are variables. The pattern $[x \rightarrow y]$ matches any new transition to normalize and the rules $loc(A, x) \rightarrow q_{refA}$ and $loc(B, y) \rightarrow q_{refB}$ merge all references to an object of the class A and an object of the class B into the states q_{refA} and q_{refB} , respectively.

The approximation rules for frames and states are built according to the principle illustrated in Figure 6. The frames representing two different calls to the method m of the class c are merged independently of the current state of the execution in which the method m is called. The set of approximation rules N is completed by giving such an approximation rule for each method of each

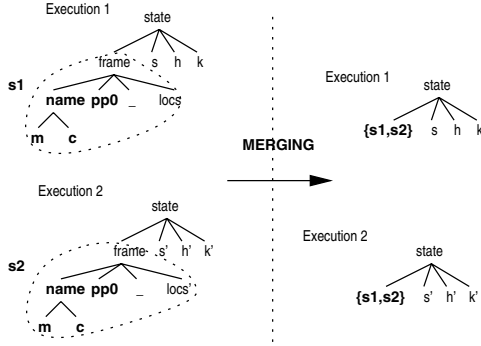


Fig. 6. Principle of approximation rules for a 0-CFA analysis

class. Using N , we can automatically obtain a fixpoint automaton $\mathcal{A}_{N,\mathcal{R}}^{145}$ over-approximating the set of all reachable Java program states. The result of the 0-CFA class analysis can be obtained, for each program location (a program point in a method in a class), by asking for the possible classes for each object in the stack or in the local variables. For instance, to obtain the set of possible classes $?c$ for the object passed as parameter to the method *execute*, i.e. the possible classes for the second local variable at program point *pp0* of *execute*, one can use the following pattern: $frame(name(execute, Test), pp0, -, locals(-, loc(?c, -), ...))$. The result obtained for this pattern is that there exist two possible values for $?c$: q_A and q_B which are the states recognizing respectively the classes *A* and *B*. This is consistent with 0-CFA which is not able to discriminate between the two possible calls to the *execute* method.

5.2 1-CFA Analysis

For 1-CFA, we need to refine the set of approximation rules into N' . In N' the rules on integers, the input stream and references are similar to the ones used for 0-CFA. In N' , approximation rules for states and frames are designed according to the principle illustrated in Figure 7. Contrary to Figure 6, the frames for the method *m* of the class *c* are merged if the corresponding method calls have been done from the same program point (in the same method m' of the class c'). For example, there are two approximation rules for the method *execute* of the class *Test*: one applying when *execute* is invoked from the program point *pp18* of the method *main*, and one applying when it is done from the program point *pp21* of this same method. Applying the same principle for all the methods, we obtain a complete set of approximation rules N' . Using N' , completion terminates on $\mathcal{A}_{N',\mathcal{R}}^{140}$. The following patterns:

```
state(frame(name(execute, Test), pp0, -, locals(-, loc(?c, -), ...)), stack(storeframe(-, pp18, ...), -)...)
state(frame(name(execute, Test), pp0, -, locals(-, loc(?c, -), ...)), stack(storeframe(-, pp21, ...), -)...)

```

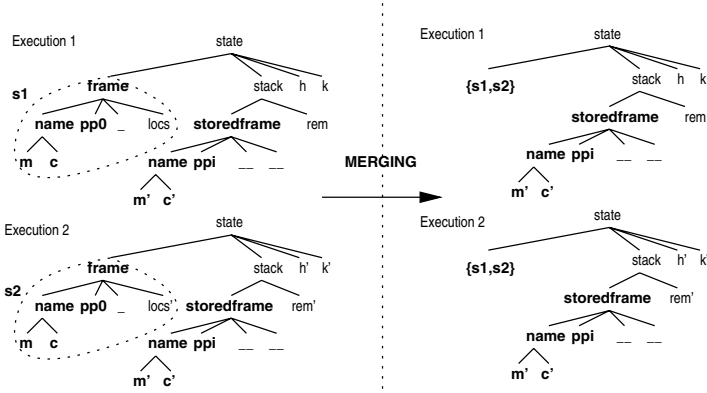


Fig. 7. Principle of approximation rules for a 1-CFA analysis

gives the desired result: each pattern has only one solution for $?c$: q_A for the first and q_B for the second. Using a similar pattern to query the 0-CFA automaton $\mathcal{A}_{N,\mathcal{R}}^{145}$, gives q_A and q_B as solution for $?c$ for both program points.

5.3 Fine-Tuning the Precision of the Analysis

Assume that we want to show that, after the execution of the previous program, field y has always a value 1 for objects of class A and 2 for objects of class B . This cannot be done by 1-CFA nor by any k -CFA since, in those analyzes, integers are abstracted by their type. One of the advantage of our technique is its ability to easily make approximation more precise by removing some approximation rules.

The property we want to prove is related to values 1 and 2 so it is tempting to refine our approximation so as not to merge those values. However, only distinguishing these two values is not enough for the analysis to succeed. Further experimentation with the approximation shows that refining the approximation of the integers by distinguishing between 0, 1, 2 and “any other integer” is enough to prove the desired property. Formally, this is expressed by the following transitions: $0 \rightarrow q_0, succ(q_0) \rightarrow q_1, succ(q_1) \rightarrow q_2, succ(q_2) \rightarrow q_{int}, succ(q_{int}) \rightarrow q_{int}$. For specifying the negative integers, the following transitions are used: $pred(q_0) \rightarrow q_{negint}$ and $pred(q_{negint}) \rightarrow q_{negint}$. The input stream representation is also modified by the following transitions: $nilstackin \rightarrow q_{in}, stackin(q_{negint}, q_{in}) \rightarrow q_{in}, stackin(q_{int}, q_{in}) \rightarrow q_{in}$ and $stackin(q_j, q_{in}) \rightarrow q_{in}$ with $j = 0, \dots, 2$.

No other approximation is needed to ensure termination of the completion. In the fixpoint automaton $\mathcal{A}_{N,\mathcal{R}}^{161}$, we are then able to show that, when the Java program terminates, there are only two possible configurations of the heap. Either the heap contains an object of class A and an object of class B whose fields are both initialized to 0, or it contains an object of class A whose field has the value 1 and an object of class B whose field has the value 2. These verifications have

been performed using a pattern matching with all the frames whose *pp* value is the last control point of the program.

This result is not surprising. The first result is possible when there is zero iterations of the loop (*x* is set to 0 before the instruction `while (x != 0){...}`). The second result is obtained for 1 or more iterations. Nevertheless, this kind of result is impossible to obtain with the two previous analyzes presented in Section 5.1 and 5.2.

6 Related Work

Term rewriting systems have been used to define and prototype semantics for a long time. However, this subject has recently reappeared for verification purposes. In [7,16], rewriting is also used as operational semantics for Java. The verification done on the obtained rewriting system is closer to finite model-checking or to simulation, since it can only deal with finite state programs. Moreover, no abstraction mechanism is proposed. Hence, our work is complementary to theirs since it permits to define abstractions in the rewrite model and to prove properties on Java applets for unbounded sets of inputs or for unbounded execution paths. In [15], abstractions on reachability analysis are defined but they seem to be too restrictive to deal with programming language semantics. Instead of tree automata, Meseguer, Palomino and Martí-Oliet use equations to define approximated equivalence classes. More precisely, they use terminating and confluent term rewriting systems normalizing every term of a class to its representative. In order to guarantee safety of approximations, approximation and specification rules must satisfy strong syntactic constraints. Roughly, approximation TRS and specification TRS, they use, have to commute. Such properties are hard to prove on a TRS encoding the Java semantics. Moreover, the approximation rules we used for class analysis are contextual and cannot easily be expressed as equations.

Takai [18] also proposed a theoretical version of approximated reachability analysis over term rewriting systems. This work also combines equations and tree automata. However, again, syntactic restrictions imposed on the equations are strong and would prevent us from constructing the kind of approximation we use on Java bytecode.

In [4], Bouajjani et al. propose a verification methodology based on abstractions and tree transducer applications on tree automata languages, called *Abstract Regular Tree Model Checking*. This brings into play a tree transducer τ , a tree automaton \mathcal{A} and an abstraction α . For a given system to verify, τ encodes its transition relation and $\mathcal{L}(\mathcal{A})$ accounts for its set of initial configurations. As for computing \mathcal{R}^* of a set of terms in rewriting, computing $\tau^*(\mathcal{L}(\mathcal{A}))$ may not terminate. A well-suited abstraction α makes the computation converge at the expense of an over-approximation of the set of configurations actually reachable. The underlying idea of this technique is close to ours. However, in our case, the TRS can implement basic computations in the semantics which would be complicated to specify in terms of transducers.

As in classical static class analyzes (such as *e.g.*, [17]), we can get several ranges of precision of k-CFA, depending of the approximation rules. In addition, starting from an automatically generated approximation, it is possible to adapt approximation rules so as to get a more precise abstraction and prove specific properties that may be difficult to show by an analyzer whose abstractions are built-in (See Section 5.3 for instance).

7 Conclusion

We have defined a technique, based on rewriting and tree automata, for prototyping static analyzers from the operational semantics of a programming language. As a test case, we showed how to produce a TRS \mathcal{R} modeling the operational semantics of a given Java program p . In this setting, given a set of inputs E the set $\mathcal{R}^*(E)$ represents the set of program states reachable by p on inputs E , i.e. the collecting semantics of p . The TRS \mathcal{R} is produced automatically and has a size linear in the size of the source program p . The technique has been implemented and experimented on a number of standard control-flow analyzes for Java bytecode, demonstrating the feasibility of the technique.

The exact set of reachable states is not computable in general so we use the tree automata completion algorithm and approximation rules so as to compute a finite approximation of the superset of $\mathcal{R}^*(E)$. The approximation technique works at the level of terms and their representation through a tree automata and has a number of advantages. First, the correctness of the approximation is guaranteed by the underlying theory and does not have to be proved for each proposed abstraction. Second, the analysis has easy access to several types of information (on integers, memory, call stacks), as illustrated in Section 5.3. This is an advantage compared to the more standard approach which combines several data flow analyzes (by techniques such as reduced and open product) to gather the same information. Third, it is relatively easy to fine-tune the analysis by adding and removing approximation rules.

We have presented our approach in terms of a sequential fragment of Java bytecode but the term rewriting setting is well suited to deal with the extension to concurrent aspects of Java and to the handling of exceptions [16,7].

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Bertelsen, P.: Semantics of Java Byte Code. Technical report, Technical University of Denmark (1997)
3. Boichut, Y., Genet, T., Jensen, T., Le Roux, L.: Rewriting Approximations for Fast Prototyping of Static Analyzers. Research Report RR 5997, INRIA (2006) <http://www.irisa.fr/lande/genet/publications.html>

4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking. In: Proceedings of 7th International Workshop on Verification of Infinite-State Systems – INFINITY 2005, in BRICS Notes Series, vol. 4, pp. 15–24 (2005)
5. 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/>
6. Dershowitz, N., Jouannaud, J.-P.: Handbook of Theoretical Computer Science. In: Rewrite Systems, vol. B(6), pp. 244–320. Elsevier, North-Holland Also as: Research report 478, LRI (1990)
7. Farzan, A., Chen, C., Meseguer, J., Rosu, G.: Formal analysis of java programs in javafan. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
8. Feuillade, G., Genet, T., Viet Triem Tong, V.: Reachability Analysis over Term Rewriting Systems. JAR 33(3-4), 341–383 (2004)
9. Freund, S.N., Mitchell, J.C.: A formal framework for the Java bytecode language and verifier. ACM SIGPLAN Notices 34(10), 147–166 (1999)
10. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) Proc. 9th RTA Conf. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)
11. Genet, T., Klay, F.: Rewriting for Cryptographic Protocol Verification. In: McAllester, D. (ed.) CADE-17, LNCS, vol. 1831, Springer, Heidelberg (2000)
12. Genet, T., Tang-Talpin, Y.-M., Viet Triem Tong, V.: Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In: Proceedings of Workshop on Issues in the Theory of Security (2003)
13. 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/>
14. Gilleron, R., Tison, S.: Regular tree languages and rewrite systems. Fundamenta Informaticae 24, 157–175 (1995)
15. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational Abstractions. In: Baader, F. (ed.) Proc. 19th CADE Conf. LNCS (LNAI), vol. 2741, pp. 2–16. Springer, Heidelberg (2003)
16. Meseguer, J., Rosu, G.: Rewriting logic semantics: From language specifications to formal analysis tools. In: IJCAR, pp. 1–44 (2004)
17. Shivers, O.: The semantics of Scheme control-flow analysis. In: Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, vol. 26, pp. 190–198, New Haven, CN, (June 1991)
18. 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)

Determining Unify-Stable Presentations

Thierry Boy de la Tour¹ and Mnacho Echenim²

¹ CNRS - Laboratoire d'Informatique de Grenoble, France

Thierry.Boy-de-la-Tour@imag.fr

² Dipartimento di Informatica

Università degli Studi di Verona, Italy

echenim@sci.univr.it

Abstract. The class of equational theories defined by so-called *unify-stable presentations* was recently introduced, as well as a complete and terminating unification algorithm modulo any such theory. However, two equivalent presentations may have a different status, one being unify-stable and the other not. The problem of deciding whether an equational theory admits a unify-stable presentation or not thus remained open. We show that this problem is decidable and that we can compute a unify-stable presentation for any theory, provided one exists. We also provide a fairly efficient algorithm for such a task, and conclude by proving that deciding whether a theory admits a unify-stable presentation and computing such a presentation are problems in the Luks equivalence class.

1 Introduction

Some equational axioms are better dealt with if kept apart from other axioms, by directly embedding them in the deduction mechanism. A typical example is reasoning modulo commutativity, which avoids having to handle an unorientable axiom. Of course, in order for this approach to be practical, the considered theory must satisfy certain requirements, and most of those concern *unification*. An equational theory a lot of research has focused on is associativity-commutativity (*AC*), and in [10], *permutative equational theories* were introduced as a generalization of *AC*. The theories in this class enjoy several interesting properties, in particular, it can be decided in polynomial time whether a theory is permutative, and the word problem for a permutative theory is decidable. However, it was proved in [14] that there exist undecidable unification problems modulo a permutative theory. This result was refined in [11] to the subclass of *variable-permuting theories*; these theories are defined by identities of the form $s \approx t$, where t is obtained from s by permuting the positions of the variables of s .

The subclass of *leaf-permutative theories* is obtained from the previous one by imposing that all the terms appearing in the presentation are linear. It is however not practical to reason modulo any leaf-permutative theory either, since there exist unification problems modulo such theories with infinite complete sets of unifiers (see, e.g., [13,9]); furthermore, to the best of our knowledge, the decidability of unification modulo a leaf-permutative theory is still an open problem.

Unify-stability. The class of leaf-permutative theories defined by a *unify-stable presentation* was introduced in [3] and a generic unification algorithm modulo any such theory was designed in [4]. This algorithm consists of a set of abstract inference rules and is thus easy to implement, it is also efficient and always returns a complete set of unifiers with at most a simply exponential cardinality. It is thus possible to perform deduction efficiently modulo a theory *presented by a unify-stable set of axioms*.

However, an equational theory can admit two equivalent presentations, only one of which is unify-stable. This is why in this paper we are interested in the following questions: given an equational theory, how do we know whether it admits a unify-stable presentation? And if such a presentation exists, how do we compute it? The first main contribution of this paper is the proof that if E defines an equational theory admitting a unify-stable presentation, then E actually *contains* such a presentation.

The Luks equivalence class. It is natural for permutation groups to arise when dealing with leaf-permutative equations, and several problems related to leaf-permutative theories can be reduced to group-theoretic problems. These group-theoretic problems have interesting complexity properties, and most of them belong to the *Luks equivalence class*, a complexity class that is close to the *graph isomorphism problem* (see, e.g., [7]). This class is included in **NP** and seems to be intermediate between **P** and **NP**-complete problems.

Efficient algorithms have been devised and implemented for the problems in this class (see, e.g., [5,6]), and can therefore be used to solve the problems on leaf-permutative theories. The second main contribution is the design of an algorithm that constructs a unify-stable presentation for a theory, if one exists, and the proof that this computation problem is in the Luks equivalence class. We also prove that the corresponding decision problem belongs to the same class.

Outline of the paper. We introduce in Section 2 a number of basic or standard definitions and notations on terms, substitutions and equational theories, and define precisely the problems we try to solve. In Section 3, after proving a few easy but necessary properties of *permutative* presentations, we introduce an order based on a notion of *ground length*, and thus solve our problems on the class of *leaf-permutative* presentations.

We then define in Section 4 the class of *unify-stable* presentations, recall some of their important properties from [4], and parallel the results of Section 3 with a *subsumption* ordering. We refine these results in Section 5 by showing that only a restricted set of terms may appear in a *minimal* presentation of a unify-stable theory. A simple algorithm thus solves our computation problems, but we provide a more subtle one in Section 6 to establish their complexity.

Due to a lack of space, some of the proofs are only outlined in this paper. A longer version, containing the complete proofs, is available in [2].

2 Preliminaries

We will use mostly standard definitions and notations, and we refer the reader to [1] for details. We build terms on a fixed infinite set of variables \mathcal{X} , and we only consider finite signatures Σ disjoint from \mathcal{X} . By Σ -terms we mean terms built on Σ and \mathcal{X} . A Σ -identity is an ordered pair of Σ -terms, denoted by $t \approx t'$; t and t' are respectively its left- and right-hand sides. It is *linear* if both sides are linear. The set of positions of a term t is denoted by $\text{Pos}(t)$, and the empty string is denoted by ε . The subterm of t at position p is denoted by $t|_p$; it is replaced by s in $t[s]_p$. A *variable position* p in t is the position of a variable $t|_p \in \mathcal{X}$.

The result of applying a substitution μ to a term t is denoted by $t\mu$, hence substitutions are composed in reverse notation: $t\mu\nu$ is the result of applying $\nu \circ \mu$ to t . A substitution ρ is a *variable renaming* if it is a permutation of \mathcal{X} . Then $t\rho$ is a *variant* of t , denoted by $t \sim t\rho$, the identity $s\rho \approx t\rho$ is a *variant* of $s \approx t$, and the substitution $\mu\rho$ is a *variant* of μ . The most general unifier of two terms s and t is denoted by $\text{mgu}(s, t)$.

A *presentation* E is a pair consisting of a (finite) signature Σ_E and a finite set of Σ_E -identities, also denoted by E . We write $E \models s \approx t$ or $s \approx_E t$ if the Σ_E -identity $s \approx t$ holds in all Σ_E -algebras in which all the identities in E are true. The relation \approx_E is the *equational theory* induced by E , and we say that E is a *presentation of this theory*. If E and F are two presentations of the same theory (i.e. they are equivalent), we may say that one is a presentation of the other. A presentation E is *minimal* if no strict subset of E is equivalent to E ; it is *standard* if distinct identities in E do not share variables.

For any class of presentations \mathcal{E} , we consider the following decision or computation problems relative to \mathcal{E} . We denote by $\text{EQF}_{\mathcal{E}}$ any function that, given as input a presentation E , returns an equivalent presentation in \mathcal{E} if one exists, and *Fail* otherwise. We denote by $\text{EQP}_{\mathcal{E}}$ the corresponding decision problem, i.e. whether the input E admits an equivalent presentation in \mathcal{E} . Finally, we consider the *uniform word problem for \mathcal{E}* , or $\text{UWP}_{\mathcal{E}}$, which takes as input a presentation $E \in \mathcal{E}$ and two Σ_E -terms s and t , and outputs the truth value of $E \models s \approx t$.

3 Permutative and Leaf-Permutative Presentations

We first establish some properties of permutative theories which we then use to design a simple decision procedure that tests whether an equational theory admits a permutative (resp. leaf-permutative) presentation or not.

Definition 1. For any Σ -term t we denote by $[t]$ the multiset of symbols occurring in t . A Σ -identity $t \approx t'$ is *permutative* if $[t] = [t']$; it is *leaf-permutative* if it is permutative, linear and $t \sim t'$. For any presentation E , the *leaf-permutative language of E* , denoted by L_E , is the set of left-hand sides of all leaf-permutative identities in E .

A presentation is (leaf-)permutative if it only contains (leaf-)permutative identities. We let \mathcal{P} (resp. \mathcal{L}) denote the class of permutative (resp. leaf-permutative) presentations.

Note that a leaf-permutative identity is always of the form $l \approx l\sigma$, where l is linear and σ is a permutation of $\text{Var}(l)$. For instance, the axiom of commutativity can be written $f(x, y) \approx f(x, y) (x y)$, where $(x y)$ is the cycle notation for the permutation $\{x \leftarrow y, y \leftarrow x\}$. The axiom of associativity $f(f(x, y), z) \approx f(x, f(y, z))$ cannot be written this way, hence is not leaf-permutative, but it is permutative since both sides are built on the same multiset of symbols $\{f, f, x, y, z\}$.

The class of permutative presentations enjoys the following stability property.

Theorem 1. *An equational theory admits a permutative presentation iff all its presentations are permutative.*

Proof. For any presentation E , by Birkhoff's Theorem, all identities $t \approx_E t'$ can be decomposed into finite sequences of reduction steps. If $E \in \mathcal{P}$ then these reduction steps preserve $[t]$, hence $[t] = [t']$, and $t \approx_E t'$ is permutative. Since all presentations of \approx_E are included in \approx_E , they are all permutative. \square

It is therefore clear that a presentation admits an equivalent permutative presentation only if it is already permutative.

Corollary 1. *$\text{EqP}_{\mathcal{P}}$ is decidable and $\text{EqF}_{\mathcal{P}}$ is computable.*

A more important and well-known (see, e.g., [10]) property is:

Theorem 2. *$\text{UWP}_{\mathcal{P}}$ is decidable.*

Proof. This is a consequence of the fact that for every $E \in \mathcal{P}$ and Σ_E -term t , the E -congruence class of t is included in the set of all Σ_E -terms t' such that $[t'] = [t]$, hence is finite. \square

Since $\mathcal{L} \subset \mathcal{P}$, Theorem 2 entails the decidability of the uniform word problem for \mathcal{L} . On the opposite, Theorem 1 does not extend to leaf-permutative presentations. For instance, to the leaf-permutative axiom of commutativity for f we can add $f(a, b) \approx f(b, a)$ (if we have constants a and b in Σ), yielding an equivalent but non leaf-permutative presentation. This presentation is of course not minimal, but this is not important here. Another example is the theory AC : its usual presentation is minimal and not leaf-permutative, yet AC admits a leaf-permutative presentation (see below).

By Theorem 1 it is obvious that a presentation E can only admit a leaf-permutative presentation if E is permutative. It is also clear that not all permutative presentations admit leaf-permutative presentations, as witnessed by the axiom of associativity. In order to search for leaf-permutative axioms in a *finite* set we introduce some new notations.

Definition 2. *Given a term t , its ground length, denoted by $|t|_{\text{g}}$, is the cardinality of the multiset $[t] \setminus \mathcal{X}$. The ground length of an identity $t \approx t'$ is defined by $|t \approx t'|_{\text{g}} = \max(|t|_{\text{g}}, |t'|_{\text{g}})$. Given a presentation E , its ground length $|E|_{\text{g}}$ is the maximal ground length of its elements. For any natural number n , we denote by $E_{\leq n}$ the subset of identities in E whose ground lengths are less than or equal to n .*

Of course in $[t] \setminus \mathcal{X}$ we discard all occurrences of variables, so that for instance, if a is a constant and $x \in \mathcal{X}$, then we have $|f(x, f(a, x))|_{\mathfrak{g}} = 3$. Note also that $|s \approx t|_{\mathfrak{g}} = |t \approx s|_{\mathfrak{g}}$, and hence $(E^{-1})_{\leq n} = (E_{\leq n})^{-1}$. We now show that identities deduced by reduction have a greater ground length than their premiss.

Lemma 1. *For all terms s, s', t, t' , if $t \rightarrow_{s \approx s'} t'$ then $|s \approx s'|_{\mathfrak{g}} \leq |t \approx t'|_{\mathfrak{g}}$.*

Proof. Since there is a position p of t and a substitution μ such that $t|_p = s\mu$ and $t' = t[s'\mu]_p$, we have

$$|s|_{\mathfrak{g}} \leq |s\mu|_{\mathfrak{g}} = |t|_p|_{\mathfrak{g}} \leq |t|_{\mathfrak{g}} \quad \text{and} \quad |s'|_{\mathfrak{g}} \leq |s'\mu|_{\mathfrak{g}} = |t'|_p|_{\mathfrak{g}} \leq |t'|_{\mathfrak{g}},$$

hence obviously $\max(|s|_{\mathfrak{g}}, |s'|_{\mathfrak{g}}) \leq \max(|t|_{\mathfrak{g}}, |t'|_{\mathfrak{g}})$. \square

The permutative identities used in a deduction can thus be bounded.

Theorem 3. *For any presentation E and presentation $F \in \mathcal{P}$, we have $F \models E$ iff $F_{\leq |E|_{\mathfrak{g}}} \models E$.*

Proof. The if part is obvious. For the only if part, consider an identity $t \approx t'$ in E . By $F \models E$ we have $t \approx_F t'$, and since $F \in \mathcal{P}$ the F -congruence class of t only contains terms s such that $[s] = [t] = [t']$. These terms all have the same ground length $n = |t \approx t'|_{\mathfrak{g}} \leq |E|_{\mathfrak{g}}$. By Birkhoff's Theorem and Lemma 1, t reduces to t' by identities in $F \cup F^{-1}$ whose ground lengths are bounded by n . Since $(F \cup F^{-1})_{\leq n} = F_{\leq n} \cup (F_{\leq n})^{-1}$ and $F_{\leq n} \subseteq F_{\leq |E|_{\mathfrak{g}}}$ we have $F_{\leq |E|_{\mathfrak{g}}} \models t \approx t'$. \square

This means that, in the search for leaf-permutative axioms for \approx_E , we need to consider only leaf-permutative identities whose sides have a ground length bounded by that of E . But since the elements of \mathcal{X} are terms of ground length 0, the set of terms of bounded ground length is always infinite. This is why we work modulo \sim .

Definition 3. *Let S and S' both denote sets of terms or sets of identities. We write $S \subsetneq S'$ if every element of S has a variant in S' . We say that S is a variant of S' , written $S \sim S'$, if $S \subsetneq S'$ and $S' \subsetneq S$. We say that S is \sim -reduced if no two distinct elements of S are variants of each other; it is obvious that for every S there is a \sim -reduced set S' such that $S \sim S'$.*

For any finite signature Σ and natural number n we let $\mathcal{T}_{\leq n}(\Sigma)$ denote a \sim -reduced set of Σ -terms that is a variant of the set of Σ -terms t such that $|t|_{\mathfrak{g}} \leq n$. Similarly, we let $\mathcal{I}_{\leq n}(\Sigma)$ denote a \sim -reduced set of Σ -identities that is a variant of the set of Σ -identities $s \approx t$ such that $|s \approx t|_{\mathfrak{g}} \leq n$.

Since Σ is finite, $\mathcal{T}_{\leq n}(\Sigma)$ and $\mathcal{I}_{\leq n}(\Sigma)$ are both finite for all n . For instance, if Σ contains a unary function symbol f , a constant a and nothing else, then we can choose for $\mathcal{T}_{\leq 2}(\Sigma)$ the set $\{x, a, f(y), f(a), f(f(x))\}$ (the particular variables used in each term are of course irrelevant). The following theorem shows that if E admits a presentation in \mathcal{L} then such a presentation can be found in $\mathcal{I}_{\leq n}(\Sigma)$.

Theorem 4. *A presentation E admits a presentation in \mathcal{L} iff $F \models E$, where*

$$F = \{t \approx t' \in \mathcal{I}_{\leq |E|_{\mathbb{g}}}(\Sigma_E) \mid t \approx t' \text{ is leaf-permutative, and } t \approx_E t'\}.$$

Proof. We have $E \models F$ and $F \in \mathcal{L}$, hence the if part is trivial. Conversely, if E admits a presentation $F' \in \mathcal{L}$, then by Theorem 3 $F'_{\leq |E|_{\mathbb{g}}}$ is also a presentation of E . Necessarily $F'_{\leq |E|_{\mathbb{g}}} \subsetneq F$, hence F is a presentation of E . \square

Corollary 2. *$\text{EQP}_{\mathcal{L}}$ is decidable and $\text{EQF}_{\mathcal{L}}$ is computable.*

Proof. If a presentation E admits a leaf-permutative presentation, then it is permutative according to Theorem 1. We thus check whether E is permutative, and if so, we can compute the finite set F of Theorem 4, since \approx_E is then decidable according to Theorem 2. Since F is permutative, \approx_F is also decidable, and we can thus decide whether $F \models E$ or not. \square

From a practical point of view, it might be more efficient to compute the leaf-permutative presentation F by enumerating the *linear* terms l in $\mathcal{T}_{\leq |E|_{\mathbb{g}}}(\Sigma_E)$. For each such l we compute its E -congruence class, and keep only the variants l' of l ; to each variant corresponds the candidate leaf-permutative identity $l \approx l'$. We also compute the F -congruence class of l , and add each new leaf-permutative identity $l \approx l'$ to F only if it does not already hold in F . We then check which yet unproven identity in E becomes true in F .

For instance, considering the usual presentation of AC , we have $|AC|_{\mathbb{g}} = 2$, and we need only consider the linear terms $l_1 = f(x, y)$, $l_2 = f(f(x, y), z)$ and $l_3 = f(x, f(y, z))$ (the linear term x can always be discarded, since it yields only a trivial leaf-permutative identity $x \approx x$). We start with $F = \emptyset$; the only identity built on l_1 which is true in AC and not in \emptyset is $f(x, y) \approx f(y, x)$, so we add it to F . Only one element of AC is not true in F : the axiom of associativity. Indeed, the F -congruence classes of its left and right-hand sides are disjoint:

$$\begin{aligned} & \{f(f(x, y), z), f(f(y, x), z), f(z, f(x, y)), f(z, f(y, x))\}, \\ & \{f(x, f(y, z)), f(x, f(z, y)), f(f(y, z), x), f(f(z, y), x)\}. \end{aligned}$$

We thus consider l_2 : its AC -congruence class contains 12 terms; half of them are variants of l_2 , two of which are in l_2 's F -congruence class. The remaining 4 terms are

$$f(f(x, z), y), f(f(z, x), y), f(f(y, z), x), f(f(z, y), x).$$

We choose to add $f(f(x, y), z) \approx f(f(x, z), y)$ to F , which suffices to merge the F -congruence classes above: $f(f(x, y), z) \approx_F f(f(y, x), z) \approx_F f(f(y, z), x)$. Hence we are done, and

$$F = \{f(x, y) \approx f(y, x), f(f(x, y), z) \approx f(f(x, z), y)\}$$

is a leaf-permutative presentation of AC .

It should be mentioned that these results are perfectly valid if we replace the ground length of Definition 2 by the standard length, i.e. the cardinality of $[t]$. However, the algorithms described above would be less efficient in the presence of constants, since they have the same length as variables. For instance, with a binary function symbol f and a constant symbol a , an enumeration 1 of linear terms of length less than that of $f(x, y)$ would include a , $f(a, a)$, $f(a, x)$, $f(x, a)$ and $f(x, y)$; with the ground length we keep only a and $f(x, y)$. Furthermore, the ground length will be necessary in the sequel, in combination with Theorem 3.

4 Unify-Stable Presentations

In this section we formally define the class of unify-stable presentations. After restating some of their properties from 4, we prove a result similar to that of Theorem 3. This result entails a first restriction on the set of identities to consider when searching for a unify-stable presentation of an equational theory, provided one exists. We shall not directly determine such a presentation, but further restrict the set of identities to consider in the following section. We start by restating definitions from 4.

Definition 4. *Given a presentation E and a linear term l , the permutation group entailed by E for l , denoted by $\Gamma_E(l)$, is the set of permutations σ of $\text{Var}(l)$ such that $l \approx_E l\sigma$.*

Given a substitution μ , we define the automorphism group of μ , denoted by $\text{Aut}(\mu)$, as the set of permutations σ of \mathcal{X} such that for every variable $x \in \mathcal{X}$ we have $x\sigma\mu \sim x\mu$.

A fundamental property of these sets is that they are permutation groups. Though infinite, the groups $\text{Aut}(\mu)$ have a rather simple structure: for instance $\text{Aut}(\{x \leftarrow f(y), y \leftarrow a\})$ contains all the permutations of $\mathcal{X} \setminus \{x, y\}$. The finite groups $\Gamma_E(l)$ are more interesting. Consider for instance the axiom

$$g(f(x, y), f(z, u)) \approx g(f(x, z), f(y, u)). \quad (1)$$

Let $l = g(f(x, y), f(z, u))$, then $\Gamma_{\mathbb{1}}(l)$ contains only two permutations: the identity and $(y z)$. But if we add the commutativity axiom for f (denoted by C_f), then $\Gamma_{\mathbb{1}+C_f}(l)$ is generated by $(y z)$, $(x y)$ and $(z u)$, and thus contains all 24 permutations of $\text{Var}(l) = \{x, y, z, u\}$; this can be checked by using the GAP system 6. If we add instead the commutativity axiom for g , we have

$$g(f(x, y), f(z, u)) \rightarrow_{C_g} g(f(z, u), f(x, y)) = g(f(x, y), f(z, u)) (x z)(y u),$$

hence the group $\Gamma_{\mathbb{1}+C_g}(l)$ is generated by $(y z)$ and $(x z)(y u)$, and contains 8 permutations.

¹ In practice we would only enumerate linear terms with at least two variables, the only ones allowing to build non-trivial identities.

Definition 5 (Definition 4.1 of [4]). A standard presentation $E \in \mathcal{L}$ is unify-stable if, for any two identities $l \approx l\sigma$ and $l' \approx l'\sigma'$ in E ,

1. if l and l' are unifiable, then $\sigma \in \text{Aut}(\text{mgu}(l, l'))$,
2. if there is a non-variable position p of l other than ε such that $l|_p$ is unifiable with some variant of l' , then there is a substitution μ such that $l'\mu = l|_p$ and $\sigma' \in \text{Aut}(\mu)$.

We let \mathcal{U} denote the class of unify-stable presentations.

Note that, for all $E \in \mathcal{U}$ and $E' \subseteq E$, we have $E' \in \mathcal{U}$. Also, it is simple to check whether a presentation $E \in \mathcal{L}$ is unify-stable or not. This involves syntactic unification between $l|_p$ and l' for all pair of terms l, l' in L_E and all positions p of l , and checking membership of substitutions σ in groups $\text{Aut}(\mu)$ (i.e. whether for all variables x in their domain, the linear terms $x\sigma\mu$ and $x\mu$ are variants). This can clearly be done in time polynomial in the length of E .

For instance it is easy to see that the usual presentation of AC is not unify-stable: the mgu of $f(x, y)$ and $f(f(x', y'), z')$ is $\mu = \{x \leftarrow f(x', y'), y \leftarrow z'\}$, and $(x \ y) \notin \text{Aut}(\mu)$ since $x\mu \not\approx y\mu$. Intuitively, this means that reducing the term $l = f(f(x', y'), z')$ at the root by means of C_f results in a term which is not a variant of l .

The presentations $(\text{II}) + C_f$ and $(\text{II}) + C_g$ defined previously are both unify-stable. If however in (II) we replace the symbol g by f , we obtain an axiom that by itself is not unify-stable. It has actually been shown in [9] that unification modulo this axiom is not finitary.

We now extract from [4] some less trivial properties of unify-stable presentations.

Theorem 5. For every $E \in \mathcal{U}$ there exists a set S of linear terms such that $L_E \subseteq S$, and

- (1) For all Σ_E -terms t, t' other than variables or constants, and such that $t \approx_E t'$, there exist a term $l \in S \setminus \mathcal{X}$, a permutation σ of $\text{Var}(l)$ and two substitutions μ, μ' such that $l\mu = t, l\mu' = t'$ and $E \models F_l \models t \approx t'$, where $F_l = \{l \approx l\sigma\} \cup \{x\sigma\mu \approx x\mu' \mid x \in \text{Var}(l)\}$.
- (2) For all $l \in S$ the E -congruence class of l is $\{l\sigma \mid \sigma \in \Gamma_E(l)\}$.

Property (1) is similar to (and stronger than) *syntacticness* of E (see [9]), and allows to use the elements of the set S for decomposing equations modulo E . Property (2) shows that these terms are also special since their E -congruence class pertains essentially to group-theoretic notions. The set S can be obtained from E as in Definition 4.2 of [4], i.e. S is the set of most general instances of any number of elements of L_E . Alternatively, if we perform basic paramodulation inferences on E , we obtain the finite saturated set $\{l \approx l\sigma \mid l \in S, \sigma \in \Gamma_E(l)\}$, from which S can be extracted. By [12] this implies a number of nice properties for E -unification (e.g. it is finitary). Hence unify-stability appears as a sufficient condition for a theory to admit a leaf-permutative presentation saturated by basic paramodulation, though it is not a necessary condition. But, as mentioned

above, membership in \mathcal{U} can be tested in polynomial time, while the saturated set, or the set of terms S , can have cardinality exponential in the size of E . We will use the following subsumption ordering to bound the leaf-permutative identities appearing in a unify-stable presentation.

Definition 6. Let t and t' be two terms. For any position p in t' , we write $t \trianglelefteq_p t'$ if there exist a substitution μ such that $t\mu = t'_{|p}$. We write $t \trianglelefteq t'$ if there exists a position $p \in \text{Pos}(t')$ such that $t \trianglelefteq_p t'$. We write $t \triangleleft t'$ if $t \trianglelefteq t'$ and $t' \not\trianglelefteq t$.

Given a presentation E and a Σ_E -term t , we let $E_{\trianglelefteq t}$ (resp. $E_{\triangleleft t}$) denote the set of identities $s \approx s'$ in E such that $s \trianglelefteq t$ or $s' \trianglelefteq t$ (resp. $s \triangleleft t$ or $s' \triangleleft t$). If $E \in \mathcal{L}$ and t is linear, we let $E_{\sim t}$ denote the set of identities $l \approx l\sigma$ in E such that $l \sim t$.

It is standard that $s \sim t$ iff $s \trianglelefteq t$ and $t \trianglelefteq s$. An obvious consequence is that $E_{\trianglelefteq s} = E_{\trianglelefteq t}$ when $s \sim t$. Also note that $s \approx s'$ is in $E_{\trianglelefteq t}$ iff $s' \approx s$ is in $(E^{-1})_{\trianglelefteq t}$, hence $(E_{\trianglelefteq t})^{-1} = (E^{-1})_{\trianglelefteq t}$. We thus obtain a result analogous to Lemma [1](#).

Lemma 2. For any presentation E and Σ_E -terms t and t' , if $t \rightarrow_E t'$ then $t \rightarrow_{E'} t'$, where $E' = E_{\trianglelefteq t} \cap E_{\trianglelefteq t'}$.

Proof. There exist an identity $s \approx s'$ in E , a position p in t and a substitution μ such that $t_{|p} = s\mu$ and $t' = t[s'\mu]_p$. We thus have $s \trianglelefteq t$ and $s' \trianglelefteq t'$, hence the identity $s \approx s'$ belongs to both $E_{\trianglelefteq t}$ and $E_{\trianglelefteq t'}$ and we get $t \rightarrow_{E'} t'$. \square

With an additional condition we get a result analogous to Theorem [3](#).

Lemma 3. For any presentation E and Σ_E -terms t, t' such that $t \approx_E t'$, if the E -congruence class of t only contains variants of t , then $E_{\trianglelefteq t} \models t \approx t'$.

Proof. By Birkhoff's Theorem there exists a finite sequence of terms t_1, \dots, t_n such that $t = t_1$, $t' = t_n$ and $t_i \rightarrow_{E \cup E^{-1}} t_{i+1}$ for all $1 \leq i \leq n-1$. By Lemma [2](#) the identity $t_i \approx t_{i+1}$ is entailed by the set $(E \cup E^{-1})_{\trianglelefteq t_i} \cap (E \cup E^{-1})_{\trianglelefteq t_{i+1}}$, which is equal to $(E \cup E^{-1})_{\trianglelefteq t}$ since the t_i 's are all in the E -congruence class of t , hence are all variants of t . We therefore have $(E \cup E^{-1})_{\trianglelefteq t} \models t \approx t'$, and since $(E \cup E^{-1})_{\trianglelefteq t} = E_{\trianglelefteq t} \cup (E_{\trianglelefteq t})^{-1}$, this yields $E_{\trianglelefteq t} \models t \approx t'$. \square

Of course this condition is rather strong, and not true for all terms, e.g. the congruence class of $f(a, b)$ modulo commutativity contains $f(b, a)$ which is not a variant of $f(a, b)$. The properties of unify-stable presentations still entail a generalization of this result to *all* terms.

Theorem 6. For any $E \in \mathcal{U}$ and pair of Σ_E -terms t, t' , we have $t \approx_E t'$ iff $E_{\trianglelefteq t} \cap E_{\trianglelefteq t'} \models t \approx t'$.

Proof. The if part is trivial. For the only if part, we consider a set S as in Theorem [5](#), and proceed by induction on t , i.e. we assume that the implication holds for all strict subterms of t .

If t is a constant or a variable, since $t \approx_E t'$ is permutative we have $t = t'$, so that $t \approx t'$ is universally true.

Otherwise, from $t \approx_E t'$ we similarly conclude that t' cannot be a constant or a variable either. Hence according to Theorem 5(1) there exist a term $l \in S \setminus \mathcal{X}$, a permutation σ of $\text{Var}(l)$ and two substitutions μ, μ' such that $l\mu = t, l\mu' = t'$ and $E \models F_l \models t \approx t'$, where

$$F_l = \{l \approx l\sigma\} \cup \{x\sigma\mu \approx x\mu' \mid x \in \text{Var}(l)\}.$$

The term l cannot be a variable, hence for every variable x of l , $x\mu$ is a strict subterm of $l\mu = t$. Since $E \models F_l$ we deduce that $x\sigma\mu \approx_E x\mu'$, and by the induction hypothesis $E_{\triangleleft x\sigma\mu} \cap E_{\triangleleft x\mu'} \models x\sigma\mu \approx x\mu'$. We also have $l \approx_E l\sigma$, and since $l \in S$ we know by Theorem 5(2) that the E -congruence class of l only contains variants of l . Therefore by Lemma 3 we get $E_{\triangleleft l} \models l \approx l\sigma$.

For every $x \in \text{Var}(l)$, since $x\sigma\mu$ and $x\mu'$ are subterms of t and t' respectively, we have $x\sigma\mu \triangleleft t$ and $x\mu' \triangleleft t'$, hence $E_{\triangleleft x\sigma\mu} \subseteq E_{\triangleleft t}$ and $E_{\triangleleft x\mu'} \subseteq E_{\triangleleft t'}$. Similarly, from $l\mu = t$ and $l\mu' = t'$ we have $l \triangleleft t$ and $l \triangleleft t'$, so that $E_{\triangleleft l} \subseteq E_{\triangleleft t} \cap E_{\triangleleft t'}$. We conclude that $E_{\triangleleft t} \cap E_{\triangleleft t'} \models F_l$, and therefore that $E_{\triangleleft t} \cap E_{\triangleleft t'} \models t \approx t'$. \square

We could directly use this result to determine a finite set of leaf-permutative identities that contains suitable axioms, as in Theorem 4. However, since unifiability results from an interaction between axioms, and not just independent properties of each axiom as for presentations in \mathcal{L} , we would need to search for suitable subsets of this finite set of identities. This is why it is desirable to determine as small a set as possible. In the next section, we further exploit the unifiability hypothesis to determine a relatively “small” set.

5 Leaf-Permutative Languages and Minimality

In this section, we prove that an equational theory defined by a presentation E admits a unifiability-stable presentation if and only if E contains such a presentation. We actually prove the stronger result that any *minimal* presentation of E must be unifiability-stable, which already entails a reasonably efficient algorithm that computes a unifiability-stable presentation for E if one exists. We will describe an optimal algorithm for this problem in the following section.

We first prove a semantic generalization of the property that every subset of a presentation in \mathcal{U} is also in \mathcal{U} .

Theorem 7. *For every standard leaf-permutative presentation E and $F \in \mathcal{U}$, if $F \models E$ and $L_E \subsetneq L_F$ then $E \in \mathcal{U}$.*

Proof (sketch). Since $F \models E$, for every linear term l we have $\Gamma_E(l) \subseteq \Gamma_F(l)$. Let S be a set of linear Σ_F -terms obtained from F according to Theorem 5. In order to prove that E is unifiability-stable, we consider two identities $l \approx l\sigma$ and $l' \approx l'\sigma'$ in E . Since l and l' are in L_E they have variants $l\rho$ and $l'\rho'$ in L_F , hence in S . Assuming either that l or $l|_p$ is unifiable with a variant of l' , we get the same property on $l\rho$ and $l'\rho'$. We can then prove membership of σ or σ' in the required automorphism group, using the inclusions $\Gamma_E(l) \subseteq \Gamma_F(l)$, Definition 3.11 of 4 on $l\rho$ and $l'\rho'$, and showing that the groups involved are consistently transformed by ρ and ρ' . \square

We can thus restrict the search for suitable leaf-permutative axioms for E to those built on the linear terms belonging to the leaf-permutative language L_F of some equivalent unify-stable presentation F , and we will see that such a language is readily available. To determine this language, we first need results combining the subsumption order with the ground length.

Lemma 4. *For any presentation $E \in \mathcal{L}$, term s and linear term l :*

- (1) if $s \trianglelefteq l$ then $|s|_g \leq |l|_g$,
- (2) if $s \trianglelefteq l$ and $|s|_g = |l|_g$ then $s \sim l$,
- (3) $|E_{\triangleleft l}|_g < |l|_g$.

Proof. (1) There is a position p in l and a substitution μ such that $s\mu = l|_p$, hence we get $|s|_g \leq |s\mu|_g = |l|_p|_g \leq |l|_g$ as in Lemma [1](#).
 (2) From $|s|_g = |l|_g$ we deduce that $p = \varepsilon$ and $|s|_g = |s\mu|_g$, hence $[s] \setminus \mathcal{X}$ and $[s\mu] \setminus \mathcal{X}$ have the same cardinality. But all occurrences in $[s]$, other than variables, are in $[s\mu]$; therefore $[s] \setminus \mathcal{X} = [s\mu] \setminus \mathcal{X}$. Hence for all variables $x \in \text{Var}(s)$, $x\mu$ must also be a variable [2](#). Since $s\mu = l$ is linear, μ must be injective from $\text{Var}(s)$ to $\text{Var}(l)$, therefore $s \sim l$.
 (3) From what precedes, if $s \triangleleft l$ then $|s|_g < |l|_g$, which clearly yields $|E_{\triangleleft l}|_g < |l|_g$. \square

Lemma 5. *For any $E \in \mathcal{L}$ and linear Σ_E -terms l and l' we have*

$$|E_{\triangleleft l} \cap E_{\triangleleft l'}|_g \leq |l \approx l'|_g.$$

The equality holds only if $l \sim l'$.

Proof. Since $E \in \mathcal{L}$, all identities $s \approx s'$ in the set $E_{\triangleleft l} \cap E_{\triangleleft l'}$ are leaf-permutative, hence $s \sim s'$, and therefore $s \trianglelefteq l$ and $s \trianglelefteq l'$. By Lemma [4](#)(1) we thus have $|s|_g = |s'|_g \leq \min(|l|_g, |l'|_g)$; therefore

$$|s \approx s'|_g = |s|_g \leq \min(|l|_g, |l'|_g) \leq \max(|l|_g, |l'|_g) = |l \approx l'|_g.$$

This proves that $|E_{\triangleleft l} \cap E_{\triangleleft l'}|_g \leq |l \approx l'|_g$.

If the equality holds there must be an identity $s \approx s'$ in $E_{\triangleleft l} \cap E_{\triangleleft l'}$ such that $|s \approx s'|_g = |l \approx l'|_g$, hence $\min(|l|_g, |l'|_g) = \max(|l|_g, |l'|_g)$, which shows that $|l|_g = |l'|_g = |s|_g$. We also have $s \trianglelefteq l$ and $s \trianglelefteq l'$; by Lemma [4](#)(2) we deduce that $s \sim l$ and $s \sim l'$, and therefore $l \sim l'$. \square

We now prove that the terms occurring in a minimal presentation must also occur in every equivalent unify-stable presentation.

Theorem 8. *If E is minimal and admits a unify-stable presentation F then $E \in \mathcal{L}$ and $L_E \lesssim L_F$.*

² This is where the notion of ground length is necessary, and cannot be replaced by the standard length. For instance, we have $f(x, y, z) \trianglelefteq f(x, y, a)$, and these linear terms have the same length, but they are not variants.

Proof. For all identities $t \approx t'$ in E we have $t \approx_F t'$; by Theorem 6 we deduce that $F' \models t \approx t'$, where $F' = F_{\triangleleft t} \cap F_{\triangleleft t'}$. The presentation E must be permutative by Theorem 1, hence by Theorem 3 $E \models F'$ iff $E_{\leq |F'|_g} \models F'$. But $E \models F'$ holds since $F' \subseteq F$, therefore $E_{\leq |F'|_g} \models t \approx t'$. In the sequel we first prove that t and t' are linear, and then that $t \sim t'$, thus showing that $t \approx t'$ is leaf-permutative and that $E \in \mathcal{L}$. We then prove that t has a variant in L_F , hence that $L_E \subsetneq L_F$.

Suppose t is not linear, then t' is not linear either, since $[t] = [t']$. Since F only contains linear identities, we deduce that $F' = F_{\triangleleft t} \cap F_{\triangleleft t'}$. Consider an identity $l \approx l\sigma$ in F' , by definition $l \triangleleft t, l \triangleleft t'$. Since $F \in \mathcal{U}$ and $l \in L_F$, by Theorem 5(2) the F -congruence class of l contains only variants of l . Since E and F are equivalent, $l \approx_E l\sigma$ holds, hence by Lemma 3 we have $E_{\triangleleft l} \models l \approx l\sigma$. Since $t \approx t'$ cannot belong to any set $E_{\triangleleft l}$ with $l \in L_{F'}$, it cannot belong to their union either, and we have just proved that

$$\bigcup_{l \in L_{F'}} E_{\triangleleft l} \models F' \models t \approx t'.$$

This contradicts the minimality of E ; the terms t and t' are therefore linear.

Suppose now that $t \not\sim t'$, then by Lemma 5 we get $|F'|_g < |t \approx t'|_g$; the identity $t \approx t'$ cannot belong to $E_{\leq |F'|_g}$. Since $E_{\leq |F'|_g} \models t \approx t'$, this contradicts the minimality of E , the identity $t \approx t'$ is therefore leaf-permutative.

We finally suppose that t has no variant in L_F . Then $F' = F_{\triangleleft t} = F_{\triangleleft t'}$, and by Lemma 4(3) $|F'|_g < |t|_g = |t \approx t'|_g$. As above this means that $t \approx t'$ does not belong to $E_{\leq |F'|_g}$, which is again impossible. \square

We can thus drastically restrict the search for leaf-permutative axioms to the ones that are already given.

Theorem 9. *A standard presentation E admits a unify-stable presentation iff every minimal presentation $E' \subseteq E$ of E is unify-stable.*

Proof. The if part is trivial, because there is always a minimal presentation of E included in E . For the only if part, suppose that $F \in \mathcal{U}$ is a presentation of E and that $E' \subseteq E$ is minimal and equivalent to E , then F is also a presentation of E' . By Theorem 8 we have $E' \in \mathcal{L}$ and $L_{E'} \subsetneq L_F$. Since E' is contained in E it is obviously standard, by Theorem 7 we get $E' \in \mathcal{U}$. \square

Corollary 3. *$\text{EqP}_{\mathcal{U}}$ is decidable and $\text{EqF}_{\mathcal{U}}$ is computable.*

Proof. We first check whether E is permutative, and if so we standardize E and then eliminate all redundant identities, i.e. the identities $s \approx t$ in E such that $E \setminus \{s \approx t\} \models s \approx t$, which is decidable according to Theorem 2. We are left with a minimal and standard presentation E' of E , and E' is an answer to the problem iff it is unify-stable, which is of course decidable. \square

For instance, we deduce that AC has no presentation in \mathcal{U} , since AC is minimal and $AC \notin \mathcal{U}$. Note that this result can also be deduced from the fact that there exist unification problems modulo AC with complete sets of unifiers of double-exponential cardinality (see [8]), while unify-stable unification is simply exponential.

```

 $F_{\mathcal{U}}(E) =$ 
   $F := \emptyset; E' := E \setminus \approx_F;$ 
  while  $\exists l \approx l\sigma \in E'$  s.t.  $E'_{\triangleleft l} = \emptyset$  do
     $F := F \cup E'_{\sim l};$ 
    if  $F \notin \mathcal{U}$  then return Fail;
     $E' := E' \setminus \approx_F;$ 
  done;
  return  $F$ 

```

Fig. 1. From \mathcal{L} to \mathcal{U}

6 Complexity

We now prove that $\text{EQP}_{\mathcal{E}}$ and $\text{EQF}_{\mathcal{E}}$ are both in the Luks equivalence class by providing an optimal algorithm that computes a unify-stable presentation for an equational theory, provided one exists. It was shown in [3] that the problem $\text{UWP}_{\mathcal{U}}$ is in the Luks equivalence class. The algorithm presented in Corollary 3 solves $\text{EQF}_{\mathcal{U}}$ by using an oracle for the uniform word problem for \mathcal{P} , but not for $\text{UWP}_{\mathcal{U}}$, and therefore does not provide a Turing-reduction from $\text{EQF}_{\mathcal{U}}$ to $\text{UWP}_{\mathcal{U}}$. For this we need the more subtle algorithm of Figure 1 that, given a presentation $E \in \mathcal{L}$, builds a unify-stable presentation for E starting from the empty set.

Lemma 6. *For any standard $E \in \mathcal{L}$, $F_{\mathcal{U}}(E)$ returns a unify-stable presentation of E if one exists, and Fail otherwise.*

Proof. It is easy to see that, each time we enter the loop, we have $F \in \mathcal{U}$, $E' \cup F \subseteq E$, $E' = E \setminus \approx_F$ and $E' \cup F \models E$ since we only remove from E' the consequences of F . It is also clear that, at each iteration at least the identity $l \approx l\sigma$ is removed from E' (because it is added to F). The iteration ends when $E' = \emptyset$ or if a failure occurs. Hence the algorithm terminates, and if it returns F , then we must have $E' = \emptyset$, otherwise the test of the while loop obviously succeeds (there is a smallest $l \in L_{E'}$ w.r.t. \triangleleft); we therefore have $F \subseteq E$, $F \models E$ and $F \in \mathcal{U}$, i.e. F is a unify-stable presentation of E .

We now assume that E admits a unify-stable presentation, and show that $F_{\mathcal{U}}(E)$ computes one of those. We need to establish another invariant for the loop: that there exists a presentation $U \in \mathcal{U}$ of E such that $F \subseteq U \subseteq E$. This is true before the first iteration according to Theorem 9, since $F = \emptyset$. Suppose now that it is true when we enter the loop, and that there is an identity $l \approx l\sigma$ in E' such that $E'_{\triangleleft l} = \emptyset$. Since $E' \subseteq E$ we have $E'_{\sim l} \subseteq E$, so that $F \cup E'_{\sim l} \subseteq U \cup E'_{\sim l} \subseteq E$. Hence obviously $U' = U \cup E'_{\sim l}$ is a presentation of E , and there only remains to show that $U' \in \mathcal{U}$.

Since $E' = E \setminus \approx_F$, we have $E'_{\triangleleft l} = E_{\triangleleft l} \setminus \approx_F = \emptyset$. Thus $E_{\triangleleft l} \subseteq \approx_F$, i.e. $F \models E_{\triangleleft l}$. From $U \subseteq E$ we deduce $U_{\triangleleft l} \subseteq E_{\triangleleft l}$, hence $F \models U_{\triangleleft l}$. But $F \not\models l \approx l\sigma$, since this identity is in E' , hence $U_{\triangleleft l} \not\models l \approx l\sigma$. However, since U is a presentation of E , we have $U \models l \approx l\sigma$ and by Theorem 6, $U_{\triangleleft l} \models l \approx l\sigma$. This proves that $U_{\triangleleft l} \setminus U_{\triangleleft l}$ cannot be empty, hence that U must contain an identity whose left-hand side is

a variant of l . Thus $L_U \sim L_U \cup \{l\} = L_{U'}$, and since $U \models U'$, by Theorem 7 we get $U' \in \mathcal{U}$.

Since $F \cup E'_{\sim l} \subseteq U'$, it is obvious by Definition 5 that this new value of F is also unify-stable, and that the value Fail cannot be returned if E admits a unify-stable presentation. \square

We can now use this algorithm to efficiently compute a function $\text{EqF}_{\mathcal{U}}$.

Theorem 10. $\text{EqF}_{\mathcal{U}}$ *polynomially reduces to* $\text{UWP}_{\mathcal{U}}$.

Proof. We first check whether the input E to $\text{EqF}_{\mathcal{U}}$ is permutative, and return Fail if it is not. If $E \in \mathcal{P}$, we standardize it and split it into a set N of non-leaf-permutative identities and a set $L \in \mathcal{L}$. According to Theorem 9, if E admits a unify-stable presentation then one is included in L , hence L and E are equivalent. We thus evaluate $F_{\mathcal{U}}(L)$ using the oracle for $\text{UWP}_{\mathcal{U}}$; if it fails, then L , and thus E have no unify-stable presentation. Otherwise, it returns a presentation $F \in \mathcal{U}$ of L . We finally test whether $F \models N$ or not, again using the oracle for $\text{UWP}_{\mathcal{U}}$. If this holds, then we return F since it is a presentation of E , otherwise we return Fail because E is not equivalent to L .

It is easy to see that this algorithm runs in polynomial time; the number of iterations in $F_{\mathcal{U}}(L)$ is bounded by $|L|$, the test $F \notin \mathcal{U}$ is polynomial in the length of $F \subseteq L$, and the oracle is used at most $|L|(|L| + 1)/2 + |N|$ times. \square

We finally prove that the complexity of this algorithm is optimal.

Theorem 11. $\text{UWP}_{\mathcal{U}}$ *reduces to* $\text{EqP}_{\mathcal{U}}$ *by a polynomial transformation.*

Proof. Starting from the input $E \in \mathcal{U}$ and Σ_E -terms t, t' of $\text{UWP}_{\mathcal{U}}$ we compute an instance E' of $\text{EqP}_{\mathcal{U}}$ by adding to a standardized E an identity $s \approx s'$, obtained from $t \approx t'$ by replacing variables by new constant symbols. This transformation is obviously polynomial, and $E \models s \approx s'$ iff $E \models t \approx t'$. Moreover, since s and s' are ground $\Sigma_{E'}$ -terms, they are variants only if they are identical, hence the identity $s \approx s'$ is either trivially true or not leaf-permutative.

Suppose E' is a positive instance of $\text{EqP}_{\mathcal{U}}$, then by Theorem 9 any minimal presentation $F \subseteq E'$ of E' is in \mathcal{U} , hence is leaf-permutative, and cannot contain $s \approx s'$. This proves that $F \subseteq E$, hence that $E \models s \approx s'$, we deduce that $E \models t \approx t'$. Conversely, if E' has no unify-stable presentation, then E cannot be a presentation of E' , hence $E \not\models s \approx s'$, and therefore $E \not\models t \approx t'$. \square

It is clear that $\text{EqP}_{\mathcal{U}}$ polynomially reduces to $\text{EqF}_{\mathcal{U}}$, we conclude:

Corollary 4. $\text{EqF}_{\mathcal{U}}$ and $\text{EqP}_{\mathcal{U}}$ *are in the Luks equivalence class.*

7 Conclusion

In this paper, we investigated the problem of deciding whether an equational theory admits a unify-stable presentation, and presented an algorithm that

solves this problem. We also proved that this problem is in the Luks equivalence class, and can therefore be solved efficiently by using powerful tools from computational group theory. This result, together with the unification algorithm of [4] make it reasonable to envisage performing deduction modulo any leaf-permutative theory under the unify-stability condition.

A direction for future work is to investigate how to define a broader class of leaf-permutative theories including that of associativity-commutativity and enjoying similar properties. Such considerations may also provide insights on other problems such as the decidability of unification in leaf-permutative theories. Another direction for future work, in the case where a presentation E is not unify-stable, is to investigate how to extract the “best” possible subset of E that is unify-stable, where the meaning of “best” remains to be defined.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Boy de la Tour, T., Echenim, M.: Determining unify-stable presentations (long version) <http://www-leibniz.imag.fr/~boydelat/papers/>
3. Boy de la Tour, T., Echenim, M.: Overlapping leaf permutative equations. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 430–444. Springer, Heidelberg (2004)
4. Boy de la Tour, T., Echenim, M.: Permutative rewriting and unification. *Information and Computation* 25(4), 624–650 (2007)
5. Butler, G. (ed.): Fundamental Algorithms for Permutation Groups. LNCS, vol. 559. Springer, Heidelberg (1991)
6. The GAP Group. GAP – Groups, Algorithms, and Programming, Version 4.4 (2006) (<http://www.gap-system.org>)
7. Hoffmann, C. (ed.): Group-Theoretic Algorithms and Graph Isomorphism. LNCS, vol. 136. Springer, Heidelberg (1982)
8. Kapur, D., Narendran, P.: Double-exponential complexity of computing a complete set of ac-unifiers. In: Scedrov, A. (ed.) LICS 1992, pp. 11–21. IEEE Computer Society Press, Washington, DC, USA (1992)
9. Kirchner, C., Klay, F.: Syntactic theories and unification. In: Mitchell, J. (ed.) LICS 1990, pp. 270–277. IEEE Computer Society Press, Washington, DC, USA (1990)
10. Lankford, D.S., Ballantyne, A.: Decision procedures for simple equational theories with permutative axioms: complete sets of permutative reductions. Technical report, Univ. of Texas at Austin, Dept. of Mathematics and Computer Science (1977)
11. Narendran, P., Otto, F.: Single versus simultaneous equational unification and equational unification for variable-permuting theories. *J. Autom. Reasoning* 19(1), 87–115 (1997)
12. Nieuwenhuis, R.: Decidability and complexity analysis by basic paramodulation. *Information and Computation* 147(1), 1–21 (1998)
13. Schmidt-Schauß, M.: Solution to problems p140 and p141. *Bulletin of the EATCS* 34, 274–275 (1988)
14. Schmidt-Schauß, M.: Unification in permutative equational theories is undecidable. *J. Symb. Comput.* 8(4), 415–421 (1989)

Confluence of Pattern-Based Calculi

Horatiu Cirstea and Germain Faure

Université Nancy 2 & Université Henri Poincaré & LORIA
BP 239, F-54506 Vandoeuvre-lès-Nancy France
first.last@loria.fr

Abstract. Different pattern calculi integrate the functional mechanisms from the λ -calculus and the matching capabilities from rewriting. Several approaches are used to obtain the confluence but in practice the proof methods share the same structure and each variation on the way pattern-abstractions are applied needs another proof of confluence.

We propose here a generic confluence proof where the way pattern-abstractions are applied is axiomatized. Intuitively, the conditions guarantee that the matching is stable by substitution and by reduction.

We show that our approach directly applies to different pattern calculi, namely the lambda calculus with patterns, the pure pattern calculus and the rewriting calculus. We also characterize a class of matching algorithms and consequently of pattern-calculi that are not confluent.

1 Introduction

Pattern matching, *i.e.* the ability to discriminate patterns is one of the main basic mechanisms human reasoning is based on. This concept is present since the beginning of information processing modeling. Instances of it can be traced back to pattern recognition and it has been extensively studied when dealing with strings [15], trees [9] or feature objects [1].

It is somewhat astonishing that one of the most commonly used models of computation, the lambda calculus, uses only trivial pattern matching. This has been extended, initially for programming concerns, either by the introduction of patterns in lambda calculi [19], or by the introduction of matching and rewrite rules in functional programming languages. There are several formalisms that address the integration of pattern matching capabilities with the lambda calculus; we can mention the λ -calculus with patterns [24], the rewriting calculus [6], the pure pattern calculus [10] and the λ -calculus with constructors [2].

Each of these pattern-based calculi differs on the way patterns are defined and on the way pattern-abstractions are applied. Thus, patterns can be simple variables like in the λ -calculus, algebraic terms like in the algebraic ρ -calculus [7], special (static) patterns that satisfy certain (semantic or syntactic) conditions like in the λ -calculus with patterns or dynamic patterns that can be instantiated and possibly reduced like in the pure pattern calculus and some versions of the ρ -calculus. The underlying matching theory strongly depends on the form of the patterns and can be syntactic, equational or more sophisticated [10,4].

Although some of these calculi just extend the λ -calculus by allowing pattern-abstractions instead of variable-abstractions the confluence of these formalisms is lost when no restrictions are imposed.

Several approaches are then used to recover confluence. One of these techniques consists in syntactically restricting the set of patterns and then showing that the reduction relation is confluent for the chosen subset. This is done for example in the λ -calculus with patterns and in the ρ -calculus (with algebraic patterns). The second technique considers a restriction of the initial reduction relation (that is, a strategy) to guarantee that the calculus is confluent on the whole set of terms. This is done for example in the pure pattern calculus where the matching algorithm is a partial function (whereas any term is a pattern).

Nevertheless we can notice that in practice, the proof methods share the same structure and that each variation on the way pattern-abstractions are applied needs another proof of confluence. There is thus a need for a more abstract and more modular approach in the same spirit as in [18,11]. A possible way to have a unified approach for proving the confluence is the application of the general and powerful results on the confluence of higher-order rewrite systems [14,17,21]. Although these results have already been applied for some particular pattern-calculi [5] the encoding seems to be rather complex for some calculi and in particular for the general setting proposed in this paper. Moreover, it would be interesting to have a framework where the expressiveness and (confluence) properties of the different pattern calculi can be compared.

In this paper, we show that all the pattern-based calculi using a unitary matching algorithm can be expressed in a general calculus parameterized by a function that defines the underlying matching algorithm and thus the way pattern-abstractions are applied. This function can be instantiated (implemented) by a unitary matching algorithm as in [6,10] but also by an anti-pattern matching algorithm [12] or it can be even more general [16]. We propose a generic confluence proof where the way pattern-abstractions are applied is axiomatized. Intuitively, the sufficient conditions to ensure confluence guarantee that the (matching) function is stable by substitution and by reduction.

We apply our approach to several classical pattern calculi, namely the λ -calculus with patterns, the pure pattern calculus and the ρ -calculus. For all these calculi, we give the encodings in the general framework and we obtain proofs of confluence. This approach does not provide confluence proofs for free but it establishes a proof methodology and isolates the key points that make the calculi confluent. It can also point out some matching algorithms that although natural at the first sight can lead to non-confluent reductions in the corresponding calculus.

Outline of the paper. In Section 2 we give the syntax and semantics of the dynamic pattern λ -calculus. The hypotheses under which the calculus is confluent and the main theorems are stated in Section 3. A non-confluent calculus is given at the end of this section. In Section 4 we give the encoding of different pattern-calculi and the corresponding confluence proofs. Section 5 concludes and gives some perspectives to this work.

$A, B ::= x$	(Variable)
$\quad c$	(Constant)
$\quad \lambda_{\theta} A.B$	(Abstraction)
$\quad A B$	(Application)

Fig. 1. Syntax of the (core) dynamic pattern λ -calculus

2 The Dynamic Pattern λ -Calculus

In this section, we first define the syntax and the operational semantics of the core dynamic pattern λ -calculus. We then give the general definition of the dynamic pattern λ -calculus.

2.1 Syntax

The syntax of the core dynamic pattern λ -calculus is defined in Figure 1. It consists of variables (denoted by x, y, z, \dots), constants (denoted a, b, c, d, e, f, \dots), abstractions and applications. In an abstraction of the form $\lambda_{\theta} A.B$ we call the term A the pattern and the term B the body. The set θ is a subset of the set of variables of A and represents the set of variables bound by the abstraction. This set is often omitted when it is exactly the set of free variables of the pattern. We sometimes use an algebraic notation $f(A_1, \dots, A_n)$ for the term $((f A_1) \dots) A_n$.

Comparing to the λ -calculus we abstract not only on variables but on general terms and the set of variables bound by an abstraction is not necessarily the same as the set of (free) variables of the corresponding pattern. We say thus that the patterns are dynamic since they can be instantiated and possibly reduced.

Definition 1 (Free and bound variables). *The set of free and bound variables of a term A , denoted $\text{fv}(A)$ and $\text{bv}(A)$, are defined inductively by:*

$$\begin{array}{llll}
 \text{fv}(c) \triangleq \emptyset & \text{bv}(c) \triangleq \emptyset & \text{fv}(x) \triangleq \{x\} & \text{bv}(x) \triangleq \emptyset \\
 \text{fv}(AB) \triangleq \text{fv}(A) \cup \text{fv}(B) & & \text{fv}(\lambda_{\theta} A.B) \triangleq (\text{fv}(A) \cup \text{fv}(B)) \setminus \theta & \\
 \text{bv}(AB) \triangleq \text{bv}(A) \cup \text{bv}(B) & & \text{bv}(\lambda_{\theta} A.B) \triangleq \text{bv}(A) \cup \text{bv}(B) \cup \theta &
 \end{array}$$

When the set of free variables of a term is empty we say that the term is closed. Otherwise, it is open.

In what follows we work modulo α -conversion, that is two terms that are α -convertible are not distinguishable. Equality modulo α -equivalence is denoted here by \equiv . We adopt Barendregt's *hygiene-convention* [3], i.e. free and bound variables have different names.

A *substitution* is a partial function from variables to terms (we use post-fix notation for substitution application). We denote by $\sigma = \{x_1 \leftarrow A_1, \dots, x_n \leftarrow A_n\}$ the substitution that maps each variable x_i to a term A_i . The set $\{x_1, \dots, x_n\}$ is called the domain of σ and is denoted by $\text{Dom}(\sigma)$. The range of a substitution σ , denoted by $\text{Ran}(\sigma)$, is the union of the sets $\text{fv}(x\sigma)$ where $x \in \text{Dom}(\sigma)$. The

composition of two substitutions σ and τ is denoted $\sigma \circ \tau$ and defined as usually, that is $x(\sigma \circ \tau) = (x\tau)\sigma$. We denote by id the empty substitution. The restriction of (the domain of) a substitution σ to a set of variables θ is denoted by $\sigma|_{\theta}$.

Definition 2 (Substitution). *The application of a substitution σ to a term A is inductively defined by*

$$\begin{aligned} x\sigma &\triangleq A && \text{if } \sigma = \{\dots, x \leftarrow A, \dots\} \\ y\sigma &\triangleq y && \text{if } y \notin \text{Dom}(\sigma) \\ (\lambda_{\theta} A_1.A_2)\sigma &\triangleq \lambda_{\theta}(A_1\sigma).(A_2\sigma) \\ (A_1A_2)\sigma &\triangleq (A_1\sigma)(A_2\sigma) \end{aligned}$$

In the abstraction case, we take the usual precautions to avoid variable captures.

2.2 Operational Semantics

The operational semantics of the core dynamic pattern λ -calculus is given by a single reduction rule that defines the way pattern-abstractions are applied. This rule, given in Figure 2, is parameterized by a partial function, denoted $\text{Sol}(A \ll_{\theta} B)$, that takes as parameters two terms A and B and a set θ of variables and returns a substitution.

In most of the cases this function corresponds to a pattern-matching algorithm but it can be even more general [16, 12].

Example 1 (Syntactic matching). We consider matching problems of the form $A \ll_{\theta} B$ and conjunctions of such problems built with the (associative, idempotent and with neutral element) operator \wedge . The symbol \mathbb{F} represents a matching problem without solution. The following set of terminating and confluent rules can be used to solve a (non-linear) syntactic matching problem:

$$\begin{aligned} A \ll_{\theta} A \wedge M &\rightarrow M \\ f(A_1, \dots, A_n) \ll_{\theta} f(A'_1, \dots, A'_n) \wedge M &\rightarrow \bigwedge_{i=1}^n (A_i \ll_{\theta} A'_i) \wedge M \\ f(A_1, \dots, A_n) \ll_{\theta} g(A'_1, \dots, A'_m) \wedge M &\rightarrow \mathbb{F} && f \neq g \\ f(A_1, \dots, A_n) \ll_{\theta} x \wedge M &\rightarrow \mathbb{F} \\ (x \ll_{\theta} A) \wedge (x \ll_{\theta} A') \wedge M &\rightarrow \mathbb{F} && A \neq A' \end{aligned}$$

We can define $\text{Sol}(A \ll_{\theta} B)$ as the function that normalizes the matching problem $A \ll_{\theta} B$ w.r.t. the above rewrite rules and according to the obtained result returns:

- nothing (*i.e.* is not defined) if $\text{fv}(A) \neq \theta$ or if the result is \mathbb{F} ,
- the substitution $\{x_i \leftarrow A_i\}_{i \in I}$ if the result is of the form $\bigwedge_{i \in I \neq \emptyset} x_i \ll_{\theta} A_i$,
- id , if the result is empty.

In Section 3.2 we analyze the confluence of the relation induced by this rule and more precisely of its compatible [3] and transitive closure.

Definition 3 (Compatible relation). *A relation $\mapsto_{\mathcal{R}}$ on the set of terms of the core dynamic pattern λ -calculus is said to be compatible if for all terms A , B and C s.t. $A \mapsto_{\mathcal{R}} B$ we have $AC \mapsto_{\mathcal{R}} BC$, $\lambda_{\theta}A.C \mapsto_{\mathcal{R}} \lambda_{\theta}B.C$, $CA \mapsto_{\mathcal{R}} CB$ and $\lambda_{\theta}C.A \mapsto_{\mathcal{R}} \lambda_{\theta}C.B$.*

(β)	$(\lambda_{\theta} A.B)C \quad \rightarrow \quad B\sigma$ where $\sigma = \mathit{Sol}(A \leftarrow_{\theta} C)$
-------------	---

Fig. 2. Operational semantics of the core dynamic pattern λ -calculus

Different instances of the core dynamic pattern λ -calculus are obtained when we give concrete definitions to Sol . For example, the λ -calculus can be seen as the core dynamic pattern λ -calculus such that

$$\sigma = \mathit{Sol}(A \leftarrow_{\theta} C) \text{ iff } A \text{ is a variable } x, \theta = \{x\} \text{ and } A\sigma \equiv C$$

Example 2 (Case branchings). Consider a pattern-based calculus with a case construct denoted using $|$ (a.k.a. a **match** operator as in functional programming languages). It can be encoded in the core dynamic pattern λ -calculus as follows:

$$(A_1 \mapsto B_1 | \dots | A_n \mapsto B_n) C \quad \triangleq \quad (\lambda_x (A_1 \hookrightarrow B_1 / \dots / A_n \hookrightarrow B_n).x) C$$

where x is a fresh variable, the symbols \hookrightarrow and $/$ are constants of the core dynamic pattern λ -calculus (infix notation) and the function Sol may be defined by

$$\mathit{Sol}((A_1 \hookrightarrow B_1 / \dots / A_n \hookrightarrow B_n) \leftarrow_x A_i \sigma) \quad \triangleq \quad \{x \leftarrow B_i \sigma\}$$

Some pattern-calculi come with additional features and cannot be expressed as instances of the core dynamic pattern λ -calculus. For example, the pure pattern calculus [10] and some versions of the rewriting calculus [7] reduce the application of a pattern-abstraction to a special term when the corresponding matching problem has not and will never have a solution. In the rewriting calculus [7] there is also a construction that aggregates terms and then distributes them over applications. These calculi as well as their encodings are briefly presented in Section 4.

We define thus the dynamic pattern λ -calculus as the core dynamic pattern λ -calculus extended by a set of rewrite rules. As for Sol this set, denoted ξ , is not made precise and can be considered as a parameter of the calculus. It can include for example some rules to reduce particular pattern-abstractions to a special constant representing a definitive matching failure or some extra rules describing the distributivity of certain symbols (like the structure operator of the ρ -calculus) over the applications.

In what follows, \mapsto_{β} denotes the compatible closure of the relation induced by the rule β and \mapsto_{β}^* denotes the transitive closure of \mapsto_{β} . Similarly, we will denote by $\mapsto_{\beta \cup \xi}$ the compatible closure of the relation induced by the rules β and ξ . $\mapsto_{\beta \cup \xi}^*$ denotes the transitive closure of $\mapsto_{\beta \cup \xi}$.

3 The Confluence of the Dynamic Pattern λ -Calculus

The calculus is not confluent when no restrictions are imposed on the function Sol . This is for example the case when we consider the decomposition of

applications containing free active variables (the term $(\lambda_x x a.x)$ ($((\lambda_y y.y) a)$) can be reduced either to $\lambda_y y.y$ or to $(\lambda_x x a.x) a$ which are not joinable) or when we deal with non-linear patterns (see Section 3.3). Nevertheless, the confluence is recovered for some specific definitions of *Sol* like the one used in Section 2.2 when defining the λ -calculus as a core dynamic pattern λ -calculus.

In this section we give some sufficient conditions that guarantee the confluence of the core dynamic pattern λ -calculus. Intuitively, the hypotheses introduced in Section 3.1 under which we prove the confluence of the calculus guarantee the coherence between *Sol* and the underlying relation of the calculus. The obtained results can then be generalized for a dynamic pattern λ -calculus with an extended set of rules ξ that satisfies some classical coherence conditions.

We use here a proof method introduced by Martin-Löf that consists in defining a so-called parallel reduction that, intuitively, can reduce all the redexes initially present in the term and that is strongly confluent (even if the one-step reduction is not) under some hypotheses.

Definition 4 (Parallel reduction). *The parallel reduction is inductively defined on the set of terms as follows:*

$$\frac{}{A \twoheadrightarrow A} \quad \frac{A \twoheadrightarrow A' \quad B \twoheadrightarrow B'}{AB \twoheadrightarrow A'B'} \quad \frac{A \twoheadrightarrow A' \quad B \twoheadrightarrow B'}{\lambda_\theta A.B \twoheadrightarrow \lambda_\theta A'.B'}$$

$$\frac{A \twoheadrightarrow A' \quad B \twoheadrightarrow B' \quad C \twoheadrightarrow C'}{(\lambda_\theta A.B)C \twoheadrightarrow B'\sigma'} \text{ IF } \sigma' \in \text{Sol}(A' \ll_\theta C')$$

Note that the parallel reduction is compatible. Moreover, we should note that this definition of parallel reduction does not coincide with the classical notion of developments. For example, if we use a *Sol* function that computes the substitution solving the matching between its two arguments (for example, like in Example 1) but without using the last rule) then we have

$$\frac{f x \twoheadrightarrow f x \quad x \twoheadrightarrow x \quad (\lambda y.f y) a \twoheadrightarrow f a}{(\lambda(f x).x)((\lambda y.f y) a) \twoheadrightarrow a}$$

The substitution $\{x \leftarrow a\}$ solves the syntactic matching between the terms $f x$ and $f a$ and thus, even if the initial term contains no head redex it can still be reduced using the parallel reduction.

We extend the definition of parallel reduction to substitutions having the same domain by setting $\sigma \twoheadrightarrow \sigma'$ if for all x in the domain of σ , we have $x\sigma \twoheadrightarrow x\sigma'$.

3.1 Stability of *Sol*

Preservation of free variables. First of all, when defining a higher-order calculus it is natural to ask that the set of free variables is preserved by reduction (some free variables can be lost but no free variables can appear during reduction). For example, the free variables of the term $(\lambda_\theta A.B)C$ should include the ones of the term $B\sigma$ with $\sigma = \text{Sol}(A \ll_\theta C)$. Thus, the substitution σ should instantiate

$\forall A, C, A', C'$	
$\mathbf{H}_0 :$	$Sol(A \ll_{\theta} C) = \sigma \implies \begin{cases} Dom(\sigma) = \theta \\ Ran(\sigma) \subseteq \mathbf{fv}(C) \cup (\mathbf{fv}(A) \setminus \theta) \end{cases}$
$\mathbf{H}_1 :$	$Sol(A \ll_{\theta} C) = \sigma \implies \begin{cases} \forall \tau, Var(\tau) \cap \theta = \emptyset, \\ Sol(A\tau \ll_{\theta} C\tau) = (\tau \circ \sigma) _{\theta} \end{cases}$
$\mathbf{H}_2 :$	$\begin{cases} Sol(A \ll_{\theta} C) = \sigma \\ A \mapsto A' \quad C \mapsto C' \end{cases} \implies \begin{cases} Sol(A' \ll_{\theta} C') = \sigma' \\ \sigma \mapsto \sigma' \end{cases}$

Fig. 3. Conditions to ensure confluence of the core dynamic pattern λ -calculus

all the variables bound (by the abstraction) in B , that is, all the variables in θ . Moreover, the free variables of σ should already be present in C or free in A . These conditions are enforced by the hypothesis \mathbf{H}_0 in Figure 3.

If we think of Sol as a unitary matching algorithm, the examples that do not verify \mathbf{H}_0 are often peculiar algorithms (for example the function that returns the substitution $\{x \leftarrow y\}$ for any problem). When considering non-unitary matching (not handled in this paper), there are several examples that do not verify \mathbf{H}_0 . For instance, the algorithms solving higher-order matching problems or matching problems in non-regular theories (e.g., such that $x \times 0 = 0$) do not verify \mathbf{H}_0 .

Stability by substitution. In the core dynamic pattern λ -calculus, when a pattern-abstraction is applied the argument may be open. One can wait for the argument to be instantiated and only then compute the corresponding substitution (if it exists) and reduce the application. On the other hand, one might not want to sequentialize the reduction but to perform the reduction as soon as possible. Nevertheless, the same result should be obtained for both reduction strategies. This is enforced by the hypothesis \mathbf{H}_1 in Figure 3.

If we consider that Sol performs a naive matching algorithm that does not take into account the variables in θ and such that $Sol(a \ll_{\theta} b)$ has no solution and $Sol(x \ll_{\theta} y) = \{x \leftarrow y\}$, then the hypothesis \mathbf{H}_1 is clearly not verified (take $\tau = \{x \leftarrow a, y \leftarrow b\}$).

Stability by reduction. When applying a pattern-abstraction, the argument may also be not fully reduced. Once again, if Sol succeeds and produces a substitution σ then subsequent reductions should not lead to a definitive failure for Sol . Moreover, the substitution that is eventually obtained should be derivable from σ . This is formally defined in hypothesis \mathbf{H}_2 .

The function Sol proposed in Example 1 does not satisfy this hypothesis. If we take $I \triangleq (\lambda y.y)$ then $Sol(f(x, x) \ll_x f(II, II)) = \{x \leftarrow II\}$ but $Sol(f(x, x) \ll_x f(II, I))$ has no solution. Similarly, this hypothesis is not satisfied by a matching algorithm that allows the decomposition of applications containing a so-called free *active* variable (i.e. a variable in applicative position). For example, we can have $Sol(xa \ll_x (\lambda y.y)a) = \{x \leftarrow \lambda y.y\}$ but $Sol(xa \ll_x a)$ has no solution for any classical (first-order) matching algorithm.

3.2 Sufficient Conditions for the Confluence

In this section, we first prove the confluence of the core dynamic pattern λ -calculus under the hypotheses $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$. The proof uses the standard techniques of parallel reduction first introduced by Tait and Martin-Löf. We first show that the reflexive and transitive closure of the parallel and one-step reductions are the same. Then we show that the parallel reduction has the diamond property and we deduce the confluence of the one-step reduction.

The three hypotheses given in the previous section are used for showing the strong confluence of the parallel reduction and in particular the Lemma 2. On the other hand, we can show that the reflexive and transitive closure of $\dashv\vdash$ is equal to \mapsto_{β} independently of the properties of Sol .

Lemma 1. *The following inclusions hold. $\mapsto_{\beta} \subseteq \dashv\vdash \subseteq \mapsto_{\beta}$.*

The next fundamental lemma and the diamond property of the parallel reduction are obtained by induction and are used for the confluence theorem.

Lemma 2 (Fundamental lemma). *For all terms C and C' and for all substitutions σ and σ' , such that $C \dashv\vdash C'$ and $\sigma \dashv\vdash \sigma'$ we have $C\sigma \dashv\vdash C'\sigma'$.*

Lemma 3 (Diamond property for $\dashv\vdash$). *The relation $\dashv\vdash$ satisfies the diamond property that is, for all terms A, B and C if $A \dashv\vdash B$ and $A \dashv\vdash C$ then there exists a term D such that $B \dashv\vdash D$ and $C \dashv\vdash D$.*

Theorem 1 (Confluence). *The core dynamic pattern λ -calculus with Sol satisfying $\mathbf{H}_0, \mathbf{H}_1$ and \mathbf{H}_2 is confluent.*

As we have already said most of the pattern-calculi extend the basic β rule (or its equivalent) by a set of rules. We will state here the conditions that should be imposed in order to prove the confluence of the dynamic pattern λ -calculus, conditions that turn out to be satisfied by most of the different calculi that can be expressed as instances of the dynamic pattern λ -calculus.

We show in this section that the confluence of extensions of the core dynamic pattern λ -calculus with an appropriate set of rules is easy to deduce using Yokouchi-Hikita's lemma 25 (see also 8).

Lemma 4 (Yokouchi-Hikita). *Let \mathcal{R} and \mathcal{S} be two relations defined on the same set \mathcal{T} of terms such that*

- \mathcal{R} is strongly normalizing and confluent,
- \mathcal{S} has the diamond property,
- for all A, B and C in \mathcal{T} such that $A \mapsto_{\mathcal{R}} B$ and $A \mapsto_{\mathcal{S}} C$ then there exists D such that $B \mapsto_{\mathcal{R}^* \mathcal{S} \mathcal{R}^*} D$ and $C \mapsto_{\mathcal{R}^*} D$ (Yokouchi-Hikita's diagram).

Then the relation $\mathcal{R}^ \mathcal{S} \mathcal{R}^*$ is confluent.*

Theorem 2. *The dynamic pattern λ -calculus is confluent when*

- Sol satisfies $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$,
- the set ξ of reduction rules is strongly normalizing and confluent,
- the relations $\dashv\vdash$ and ξ satisfy Yokouchi-Hikita's diagram.

Proof. Notice that $\mapsto_{\beta \cup \xi}$ and $\mapsto_{\xi} \dashv\vdash \mapsto_{\xi}$ are equal (consequence of Lemma 1) and apply Yokouchi-Hikita's lemma with $\dashv\vdash$ and the relation induced by ξ . \square

3.3 Encoding Klop's Counter-Example Using Linear Patterns

The different results we give below may seem to be limited because the conditions imposed on the matching algorithm are strong. Nevertheless, these conditions are respected by most of the pattern-calculi we have explored and relaxing them leads to classical counter-examples for the confluence.

For example, if the matching can be performed on active variables then non-confluent reductions can be obtained in both the lambda-calculus with patterns [24] and in the rewriting calculus [7]. Similarly, non-linear patterns lead to non-confluent reductions that are variations of the Klop's counter-example [13] for higher-order systems dealing with non-linear matching. This is why in the λ -calculus with patterns or in the ρ -calculus we only consider *linear* patterns and in the pure pattern calculus matching against non-linear terms always fails.

When using dynamic patterns containing variables that are not bound in the abstraction the confluence hypotheses should be carefully verified. More precisely, the behavior of non-linear rules can be encoded using *linear* and dynamic patterns. Consequently, Klop's counter-example can be encoded in the corresponding calculus that is therefore non-confluent.

Proposition 1 (Non-confluence). *The core dynamic pattern λ -calculus with Sol such that for all terms A and B and for some constant d*

$$\begin{aligned} Sol(A \leftarrow_{\emptyset} A) &= \text{id} \\ Sol(x \leftarrow_x A) &= \{x \leftarrow A\} \\ Sol(d(x, y) \leftarrow_{x, y} d(A, B)) &= \{x \leftarrow A, y \leftarrow B\} \end{aligned} \quad \textit{is not confluent.}$$

Proof. It is sufficient to remark that we can encode non-linear patterns using dynamic linear patterns as follows:

$$\lambda_x(dx x).\clubsuit \quad \triangleq \quad \lambda_{x, y}(dx y).(\lambda_{\emptyset} x.\clubsuit)y$$

where \clubsuit denotes an arbitrary term. Then we adapt the encoding of Klop's counter-example in the core dynamic pattern λ -calculus. \square

As a consequence we obtain that any pattern-calculus defined using a function Sol that satisfies the conditions in Proposition 1 is not confluent. This is somewhat surprising since the last two computations are clearly satisfied by any classical syntactic matching algorithm and the first one seems to be a reasonable choice. On the other hand, ignoring the information given by the set θ of bound variables when performing matching can lead to strange behaviors and in particular it allows for an encoding of the equality of terms.

The conditions of Proposition 1 are not satisfied by the specific calculi considered in this paper. More precisely, $Sol(A \leftarrow_{\emptyset} A)$ does not return the identity for all terms but only for a subset that is defined either statically (syntactical restrictions on patterns) like in the λ -calculus with patterns or the rewriting calculus or dynamically (only a subset of terms can be matched) like in the pure pattern calculus. In all these cases the matchable terms from the corresponding subsets cannot be used to encode Klop's counter-example.

$(\beta_{\lambda_P}) \quad (\lambda A.B)(A\sigma) \quad \rightarrow \quad B\sigma$
--

Fig. 4. Operational semantics of the λ -calculus with patterns

4 Instantiations of the Dynamic Pattern λ -Calculus

In this section, we give some instantiations of the dynamic pattern λ -calculus. For the sake of simplicity, we only give the key points for each of the pattern based calculi. All these calculi have been proved confluent under appropriate conditions; we give here confluence proofs based on these conditions and using the general confluence proof for the dynamic pattern λ -calculus. This latter approach does not provide confluence proofs for free but gives a proof methodology that focuses on the fundamental properties of the underlying matching that can be thus seen as the key issue of pattern based calculi.

4.1 λ -Calculus with Patterns

The λ -calculus with patterns was introduced in [24] as an extension of the λ -calculus with pattern-matching facilities. The set of terms is parameterized by a set of patterns Φ on which we can abstract.

The syntax of the λ -calculus with patterns is thus the one of the core dynamic pattern λ -calculus but where patterns are taken in a given set and abstractions always bind all the (free) pattern variables. Its operational semantics is given by the rule in Figure 4.

Instead of considering syntactical restrictions, we can equivalently consider that a matching problem $A \ll_{\theta} A\sigma$ has a solution only when $A \in \Phi$. The λ -calculus with patterns can thus be seen as an instance of the core dynamic pattern λ -calculus.

The calculus is not confluent (see [24] Ex. 4.18) in general but some restrictions can be imposed on the set of patterns to recover confluence. This restriction is called the rigid pattern condition (RPC) and can be defined using the parallel reduction of the calculus. The definition allows for patterns which are extensionally but not intensionally rigid, such as Ωxy with $\Omega = (\lambda x.xx)(\lambda x.xx)$. We choose a (less general) syntactical characterization [24] that excludes these pathological cases.

Definition 5 (RPC). *The set of terms satisfying RPC is the set of all terms of the λ -calculus which*

- are linear (each free variable occur at most once),
- are in normal form,
- have no active variables (i.e., no sub-terms of the form xA where x is free).

Note that reformulating **H₂** in the particular case of a *Sol* function that performs matching on closed patterns leads to a condition close to the original RPC

(the parallel reduction used in the definition of RPC is slightly different). Nevertheless, the hypothesis \mathbf{H}_2 allows the matching to be performed on patterns that are not in normal form (or not reducible to themselves) while this is not the case for the RPC.

Example 3. Pairs and projections can be encoded in the λ -calculus with patterns by directly matching on the pair encoding.

$$\begin{aligned} (\lambda(\lambda z.(z x) y).x)(\lambda z.(z A) B) &\mapsto_{\beta_{\lambda P}} x\{x \leftarrow A, y \leftarrow B\} \\ &\equiv A \end{aligned}$$

Proposition 2. *The λ -calculus with patterns is confluent if the patterns are taken in the set defined by the RPC.*

Proof. The hypotheses \mathbf{H}_0 and \mathbf{H}_1 follow immediately. To prove \mathbf{H}_2 , we can remark that if $P\sigma \dashrightarrow B$ with $P \in \text{RPC}$ then $\exists B', \sigma'$ s.t. $B' \equiv P\sigma'$ with $\sigma \dashrightarrow \sigma'$. This proves that a redex cannot overlap with P in $P\sigma$ if $P \in \text{RPC}$ and thus that the condition \mathbf{H}_2 is satisfied. We conclude the proof by applying Thm. [11](#). \square

4.2 Rewriting Calculus

The rewriting calculus was introduced in [\[6\]](#) to make explicit all the ingredients of rewriting. There are several versions of the calculus but we focus here on the ρ_{\rightarrow} -calculus [\[7\]](#) that uses a left-distributive structure operator and a special constant representing matching failures.

Note that the syntax used here is slightly different from the original presentations that introduce matching constraints. Nevertheless, the use of matching constraints is crucial only when considering explicit matching and substitution application. We also omit the type related aspects of this calculi and consider a syntactic matching (in the original version, the reduction rules of ρ_{\rightarrow} -calculus are parameterized by a matching theory \mathbb{T} but the confluence is proved for the syntactic version). The syntax of the ρ_{\rightarrow} -calculus is given in Figure [5](#).

The *patterns* are linear algebraic terms (*i.e.* terms constructed only with variables, constants and application). A *structure* is a collection of terms that can be seen either as a set of rewrite rules or as a set of results. The symbol `stk` can be considered as the special constant representing a redex whose underlying matching problem is unsolvable.

We consider a superposition relation $\not\sqsubseteq$ between (patterns and) terms whose aim is to characterize a broad class of matching equations that have not and will never have a solution (*i.e.* independently of subsequent instantiations and reductions). Thus, if $P \not\sqsubseteq A$ then $\forall \sigma_1, \sigma_2, \forall A', A\sigma_1 \mapsto A' \Rightarrow P\sigma_2 \not\equiv A'$. This definition is clearly undecidable but syntactic characterizations can be given [\[7\]](#).

The ρ_{\rightarrow} -calculus handles uniformly matching failures and eliminates them when they are not significant for the computation. The semantics of the ρ_{\rightarrow} -calculus is given in Figure [5](#).

We can consider `!` and `stk` as constants of the dynamic pattern λ -calculus and thus we can see the syntax of the ρ_{\rightarrow} -calculus as an instance of the syntax of the

<u>Syntax of the ρ_{\rightarrow}-calculus</u>			
Patterns	$P ::= x \mid \text{stk} \mid c(P, \dots, P)$	<i>(variables occur only once in any P)</i>	
Terms	$A ::= c \mid \text{stk} \mid x \mid \lambda P.A \mid A A \mid A \wr A$		
<u>Semantics of the ρ_{\rightarrow}-calculus</u>			
$(\beta_{\rho_{\text{stk}}})$	$(\lambda P.A) B \rightarrow A\sigma$	with $P\sigma = B$	
(δ)	$(A \wr B) C \rightarrow A C \wr B C$		
(stk)	$(\lambda P.A) B \rightarrow \text{stk}$	if $P \not\sqsubseteq B$	
(stk)	$\text{stk} \wr A \rightarrow A$		
(stk)	$A \wr \text{stk} \rightarrow A$		
(stk)	$\text{stk} A \rightarrow \text{stk}$		

Fig. 5. The ρ_{\rightarrow} -calculus

dynamic pattern λ -calculus. The rule $(\beta_{\rho_{\text{stk}}})$ can be considered as an instance of the (β) rule of the dynamic pattern λ -calculus.

$$(\beta_{\rho}) \quad (\lambda P.A)B \rightarrow A\sigma \quad \text{if } \sigma = \text{Sol}(P \ll B)$$

where $\text{Sol}(P \ll B)$ has a solution only when P is a pattern. Since patterns are linear algebraic terms then the Sol function can be implemented using first-order linear matching *à la* Huet (see Example [II](#)).

Proposition 3. *The ρ_{\rightarrow} -calculus with linear algebraic patterns is confluent.*

Proof. The patterns of the ρ_{\rightarrow} -calculus satisfy the RPC. The confluence of the $(\beta_{\rho_{\text{stk}}})$ rule is thus obtained as in Prop. [2](#). The relation $\delta \cup \text{stk}$ induces a terminating relation. It is also locally confluent (all critical pairs are joinable). Moreover, the relations $\dashv\vdash_{\beta_{\rho_{\text{stk}}}}$ and $(\delta \cup \text{stk})$ satisfy the Yokouchi-Hikita's diagram (easy induction of the structure of terms). Thm. [2](#) concludes the proof. \square

4.3 Pure Pattern Calculus

In the λ -calculus, data structures such as pairs of lists can be encoded. Although the λ -calculus supports some functions that act uniformly on data structures, it cannot support operations that exploit characteristics common to all data structures such as an update function that traverses any data structures to update its atoms. In the pure pattern calculus [10](#) where any term can be a pattern the focus is put on the dynamic matching of data structures.

The syntax of the pure pattern calculus is the same as the one of the dynamic pattern λ -calculus (except that the pure pattern calculus defines a single constructor).

Pattern-abstractions are applied using a particular matching algorithm. Although the original paper uses a single rule to describe application of pattern-abstractions we present it here using two rules. First, a rule that is an instance of

<u>ϕ-data structures and ϕ-matchable forms</u>			
	D	$::=$	$x (x \in \phi) \mid c \mid DA$
	E	$::=$	$D \mid \lambda_{\theta} A.B$
	where A and B are arbitrary terms		
<u>Semantics of the pure pattern calculus</u>			
(β_{pc})	$(\lambda_{\theta} A.B)C$	\rightarrow	$B\sigma$ if $\sigma = \text{Sol}(A \ll_{\theta} C)$
$(\beta_{pc}^{\text{stk}})$	$(\lambda_{\theta} A.B)C$	\rightarrow	$\lambda x.x$ if $\text{none} = \text{Sol}(A \ll_{\theta} C)$

Fig. 6. The pure pattern calculus

the (β) rule of the dynamic pattern λ -calculus. Secondly, a rule that reduces the corresponding pattern-abstraction application to the identity (the motivation for this second rule is given in [10]) when the pattern-matching does not succeed.

The matching algorithm of the pure pattern calculus is based on the notions of ϕ -data structures (denoted D) and ϕ -matchable forms (denoted E) that are given in Figure 6.

The operational semantics of the pure pattern calculus is given in Figure 6 where the partial function Sol is defined by the following equations that are applied respecting the order below

$$\begin{aligned}
 \text{Sol}(x \ll_{\theta} A) &= \{x \leftarrow A\} && \text{if } x \in \theta \\
 \text{Sol}(c \ll_{\theta} c) &= \text{id} \\
 \text{Sol}(A_1 A_2 \ll_{\theta} B_1 B_2) &= \text{Sol}(A_1 \ll_{\theta} B_1) \uplus \text{Sol}(A_2 \ll_{\theta} B_2) && \text{if } A_1 A_2 \text{ is a } \theta\text{-data structure} \\
 &&& \text{if } B_1 B_2 \text{ is a data structure} \\
 \text{Sol}(A_1 \ll_{\theta} B_1) &= \text{none} && \text{if } A_1 \text{ is a } \theta\text{-matchable form} \\
 &&& \text{if } B_1 \text{ is a matchable form}
 \end{aligned}$$

Note that the union \uplus is only defined for substitutions of disjoint domains and that the union of none and σ is always none.

At first sight, the matching algorithm may seem surprising because one decomposes application syntactically whereas it is a higher-order symbol. This is sound because the decomposition is done only on data-structures, which consist of head normal forms.

Example 4. [10] Define $\text{elim} \triangleq \lambda x. \lambda y. (xy).y$ to be the generic eliminator. For example, suppose given two constants Cons and Nil representing list constructs. We define the function $\text{singleton} \triangleq \lambda x. (\text{Cons } x \text{ Nil})$ and we check that

$$\begin{aligned}
 \text{elim singleton} &\equiv (\lambda x. \lambda y. (xy).y) (\lambda x. (\text{Cons } x \text{ Nil})) \\
 &\mapsto_{\rho_{ppc}} \lambda (\text{Cons } y \text{ Nil}).y
 \end{aligned}$$

Proposition 4. *The pure pattern calculus is confluent.*

Proof. The hypothesis \mathbf{H}_0 is true. The hypotheses \mathbf{H}_1 and \mathbf{H}_2 are not surprisingly intermediate results in [10]. In particular, Lemma 7 [10] states that the function Sol is stable by substitution and Lemma 8 [10] proves that the function Sol is stable by reduction (when it returns a substitution and when it returns none). We use this property to prove that the relation (β_{pc}^{stk}) is locally confluent (simple induction). It is trivially terminating, and thus confluent. The fact that the relations $\dashv\vdash_{\beta_{pc}}$ and (β_{pc}^{stk}) verify Yokouchi-Hikita’s diagram is obtained again by a simple induction. The only interesting case is for the term $(\lambda A_0.A_1)A_2$ but it is easy to conclude using the stability by reduction of the function Sol . \square

5 Conclusions and Future Work

We propose here a different formulation for different pattern-based calculi and we use the confluence properties of the general formalism to give alternative confluence proofs for these calculi. The general confluence proof uses the standard techniques of parallel reduction of Tait and Martin-Löf and Yokouchi-Hikita’s lemma. The method proposed by M. Takahashi [20] gives in general more elegant and shorter proofs by using the notion of developments. Reformulating the hypotheses $\mathbf{H}_0, \mathbf{H}_1$ and \mathbf{H}_2 and adapting the proofs of this paper is easy but given a pattern-calculus the reformulated hypotheses are often more difficult to prove than for the original case.

Moreover, we show that the proof of confluence of the ρ -calculus is easy to deduce from our general result as soon as the structure operator has no equational theory. Nevertheless, if one wants to switch to non-unitary matching (and this is very useful in practice [23,22]) then the β rule should return a collection of results and the structure operator should be (at least) associative and commutative. This extension is syntactically and semantically non-trivial and opens new challenging problems. Nevertheless, we think that this work is a good starting point for the study of the confluence properties of pattern-calculi with non-unitary matching.

Acknowledgments. We would like to thank C. Kirchner for useful interactions and comments on this work and D. Kesner for the many discussions we have been having on pattern-calculi.

References

1. Ait-Kaci, H., Podelski, A., Smolka, G.: A feature constraint system for logic programming with entailment. *Theoretical Computer Science* 122(1-2), 263–283 (1994)
2. Arbisser, A., Miquel, A., Ríos, A.: A lambda-calculus with constructors. In: Pfenning, F. (ed.) *RTA 2006*. LNCS, vol. 4098, pp. 181–196. Springer, Heidelberg (2006)
3. Barendregt, H.: *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. North Holland, 2nd edn. (1984)

4. Barthe, G., Cirstea, H., Kirchner, C., Liquori, L.: Pure Patterns Type Systems. In: 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2003, New Orleans, USA, pp. 250–261. ACM, New York (2003)
5. Bertolissi, C., Kirchner, C.: The rewriting calculus as a combinatory reduction system. In: Siedl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, Springer, Heidelberg (2007)
6. Cirstea, H., Kirchner, C.: The rewriting calculus — Part I *and* II. Logic Journal of the Interest Group in Pure. and Applied Logics 9, 427–498 (2001)
7. Cirstea, H., Liquori, L., Wack, B.: Rewriting calculus with fixpoints: Untyped and first-order systems. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 147–161. Springer, Heidelberg (2004)
8. Curien, P.-L., Hardin, T., Lévy, J.-J.: Confluence properties of weak and strong calculi of explicit substitutions. Journal of the ACM (JACM) 43(2), 362–397 (1996)
9. Hoffmann, C.M., O’Donnell, M.J.: Pattern matching in trees. Journal of the ACM 29(1), 68–95 (1982)
10. Jay, B., Kesner, D.: Pure pattern calculus. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, pp. 100–114. Springer, Heidelberg (2006)
11. Kesner, D.: Confluence of extensional and non-extensional lambda-calculi with explicit substitutions. Theoretical Computer Science 238(1-2), 183–220 (2000)
12. Kirchner, C., Kopetz, R., Moreau, P.-E.: Anti-pattern matching. In: European Symposium on Programming – ESOP 2007, LNCS, Braga, Portugal, Springer, Heidelberg (2007)
13. Klop, J.W.: Combinatory Reduction Systems. Ph.D. thesis, Mathematisch Centrum, Amsterdam (1980)
14. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: Introduction and survey. Theoretical Computer Science 121, 279–308 (1993)
15. Knuth, D.E., Morris, J., Pratt, V.: Fast pattern matching in strings. SIAM Journal of Computing 6(2), 323–350 (1977)
16. Liquori, L., Honsell, F., Lenisa, M.: A framework for defining logical frameworks (2007) (Submitted)
17. Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. Theoretical Comput. Sci. 192, 3–29 (1998)
18. Melliès, P.-A.: Axiomatic rewriting theory VI residual theory revisited. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 24–50. Springer, Heidelberg (2002)
19. Peyton-Jones, S.: The implementation of functional programming languages. Prentice Hall, Inc, Englewood Cliffs (1987)
20. Takahashi, M.: Parallel reductions in λ -calculus. Information and Computation 118, 120–127 (1995)
21. Terese: Term Rewriting Systems. Cambridge University Press, Cambridge (2002)
22. The Maude Team. The Maude Home Page <http://maude.cs.uiuc.edu/>
23. The Tom Team. The Tom language <http://tom.loria.fr/>
24. van Oostrom, V.: Lambda calculus with patterns. Technical report, Vrije Universiteit, Amsterdam (November 1990)
25. Yokouchi, H., Hikita, T.: A rewriting system for categorical combinators with multiple arguments. SIAM Journal on Computing 19(1), 78–97 (1990)

A Simple Proof That Super-Consistency Implies Cut Elimination

Gilles Dowek¹ and Olivier Hermant²

¹ École polytechnique and INRIA
LIX, École polytechnique, 91128 Palaiseau Cedex, France
gilles.dowek@polytechnique.edu

<http://www.lix.polytechnique.fr/~dowek>

² PPS, Université Denis Diderot
2 place Jussieu, 75251 Paris Cedex 05, France
hermant@pps.jussieu.fr

<http://www.pps.jussieu.fr/~hermant>

Abstract. We give a simple and direct proof that super-consistency implies cut elimination in deduction modulo. This proof can be seen as a simplification of the proof that super-consistency implies proof normalization. It also takes ideas from the semantic proofs of cut elimination that proceed by proving the completeness of the cut free calculus. In particular, it gives a generalization, to all super-consistent theories, of the notion of V-complex, introduced in the semantic cut elimination proofs for simple type theory.

1 Introduction

Deduction modulo is an extension of predicate logic where some axioms may be replaced by rewrite rules. For instance, the axiom $x + 0 = x$ may be replaced by the rewrite rule $x + 0 \longrightarrow x$ and the axiom $x \subseteq y \Leftrightarrow \forall z (z \in x \Rightarrow z \in y)$ by the rewrite rule $x \subseteq y \longrightarrow \forall z (z \in x \Rightarrow z \in y)$.

In the model theory of deduction modulo, it is important to distinguish the fact that some propositions are computationally equivalent, *i.e.* congruent (*e.g.* $x \subseteq y$ and $\forall z (z \in x \Rightarrow z \in y)$), in which case they should have the same value in a model, from the fact that they are provably equivalent, in which case they may have different values. This has lead, in [4], to introduce a generalization of Heyting algebras called *truth values algebras* and a notion of \mathcal{B} -valued model, where \mathcal{B} is a truth values algebra. We have called *super-consistent* the theories that have a \mathcal{B} -valued model for all truth values algebras \mathcal{B} and we have given examples of consistent theories that are not super-consistent.

In deduction modulo, there are theories for which there exists proofs that do not normalize. But, we have proved in [4] that all proofs normalize in all super-consistent theories. This proof proceeds by observing that reducibility candidates [8] can be structured in a truth values algebra and thus that super-consistent theories have reducibility candidate valued models. Then, the existence of such a model implies proof normalization [7] and hence cut elimination. As many

theories, in particular arithmetic and simple type theory, are super-consistent, we get Gentzen's and Girard's theorems as corollaries.

This paper is an attempt to simplify this proof replacing the algebra of reducibility candidates \mathcal{C} by a simpler truth values algebra \mathcal{S} . Reducibility candidates are sets of proofs. We show that we can replace each proof of such a set by its conclusion, obtaining this way sets of sequents, rather than sets of proofs, for truth values.

Although the truth values of our model are sets of sequents, our cut elimination proof uses another truth values algebra whose elements are sets of contexts: the algebra of contexts Ω , that happens to be a Heyting algebra. Besides an \mathcal{S} -valued model we build, for each super-consistent theory, an Ω -valued model verifying some properties, but this requires to enlarge the domain of the model using a technique of *hybridization*. The elements of such a hybrid model are quite similar to the V-complexes used in the semantic proofs of cut elimination for simple type theory [12,13,11,3,9]. Thus, we show that these proofs can be simplified using an alternative notion of V-complex and also that the V-complexes introduced for proving cut elimination of simple type theory can be used for other theories as well. This hybridization technique gives a proof that uses ideas taken from both methods used to prove cut elimination: normalization and completeness of the cut free calculus. From the first, come the ideas of truth values algebra and neutral proofs and from the second the idea of building a model such that sequents valid in this model have cut free proofs.

2 Super-Consistency

To keep the paper self contained, we recall in this section the definition of deduction modulo, truth values algebras, \mathcal{B} -valued models and super-consistency. A more detailed presentation can be found in [4].

2.1 Deduction Modulo

Deduction modulo [6,7] is an extension of predicate logic (either single-sorted or many-sorted predicate logic) where a theory is defined by a set of axioms Γ and a congruence \equiv , itself defined by a confluent rewrite system rewriting terms to terms and atomic propositions to propositions.

In this paper we consider natural deduction rules. These rules are modified to take the congruence \equiv into account. For example, the elimination rule of the implication is not formulated as usual

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

but as

$$\frac{\Gamma \vdash C \quad \Gamma \vdash A}{\Gamma \vdash B} C \equiv A \Rightarrow B$$

All the deduction rules are modified in a similar way, see, for instance, [7] for a complete presentation.

In deduction modulo, there are theories for which there exists proofs that do not normalize. For instance, in the theory formed with the rewrite rule $P \longrightarrow (P \Rightarrow Q)$, the proposition Q has a proof

$$\frac{\frac{\frac{\overline{P \vdash P \Rightarrow Q} \text{ axiom} \quad \overline{P \vdash P} \text{ axiom}}{P \vdash Q} \Rightarrow\text{-intro} \quad \frac{\overline{P \vdash P \Rightarrow Q} \text{ axiom} \quad \overline{P \vdash P} \text{ axiom}}{P \vdash Q} \Rightarrow\text{-elim}}{\vdash P \Rightarrow Q} \Rightarrow\text{-intro} \quad \frac{\overline{P \vdash P \Rightarrow Q} \text{ axiom} \quad \overline{P \vdash P} \text{ axiom}}{P \vdash Q} \Rightarrow\text{-elim}}{\vdash Q} \Rightarrow\text{-elim}$$

that does not normalize. In some other theories, such as the theory formed with the rewrite rule $P \longrightarrow (Q \Rightarrow P)$, all proofs strongly normalize.

In deduction modulo, like in predicate logic, closed normal proofs always end with an introduction rule. Thus, if a theory can be expressed in deduction modulo with rewrite rules only, *i.e.* with no axioms, in such a way that proofs modulo these rewrite rules strongly normalize, then the theory is consistent, it has the disjunction property and the witness property, and various proof search methods for this theory are complete.

Many theories can be expressed in deduction modulo with rewrite rules only, in particular arithmetic and simple type theory, and the notion of cut of deduction modulo subsumes the notions of cut defined for each of these theories. For instance, simple type theory can be defined as follows.

Definition 1 (Simple type theory [2,5,3]). *The sorts are inductively defined by*

- ι and o are sorts,
- if T and U are sorts then $T \rightarrow U$ is a sort.

The language contains the constants $S_{T,U,V}$ of sort $(T \rightarrow U \rightarrow V) \rightarrow (T \rightarrow U) \rightarrow T \rightarrow V$, $K_{T,U}$ of sort $T \rightarrow U \rightarrow T$, $\dot{\top}$ of sort o and $\dot{\perp}$ of sort o , $\dot{\Rightarrow}$, $\dot{\wedge}$ and $\dot{\vee}$ of sort $o \rightarrow o \rightarrow o$, $\dot{\forall}_T$ and $\dot{\exists}_T$ of sort $(T \rightarrow o) \rightarrow o$, the function symbols $\alpha_{T,U}$ of rank $\langle T \rightarrow U, T, U \rangle$ and the predicate symbol ε of rank $\langle o \rangle$.

The rules are

$$\begin{aligned} \alpha(\alpha(S_{T,U,V}, x), y), z) &\longrightarrow \alpha(\alpha(x, z), \alpha(y, z)) \\ \alpha(\alpha(K_{T,U}, x), y) &\longrightarrow x \\ \varepsilon(\dot{\top}) &\longrightarrow \top \\ \varepsilon(\dot{\perp}) &\longrightarrow \perp \\ \varepsilon(\alpha(\dot{\Rightarrow}, x), y) &\longrightarrow \varepsilon(x) \Rightarrow \varepsilon(y) \\ \varepsilon(\alpha(\dot{\wedge}, x), y) &\longrightarrow \varepsilon(x) \wedge \varepsilon(y) \\ \varepsilon(\alpha(\dot{\vee}, x), y) &\longrightarrow \varepsilon(x) \vee \varepsilon(y) \\ \varepsilon(\alpha(\dot{\forall}_T, x)) &\longrightarrow \forall y \varepsilon(\alpha(x, y)) \\ \varepsilon(\alpha(\dot{\exists}_T, x)) &\longrightarrow \exists y \varepsilon(\alpha(x, y)) \end{aligned}$$

2.2 Truth Values Algebras

Definition 2 (Truth values algebra). Let B be a set, whose elements are called truth values, B^+ be a subset of B , whose elements are called positive truth values, \mathcal{A} and \mathcal{E} be subsets of $\wp(B)$, $\tilde{\top}$ and $\tilde{\perp}$ be elements of B , \Rightarrow , $\tilde{\wedge}$, and $\tilde{\vee}$ be functions from $B \times B$ to B , $\tilde{\forall}$ be a function from \mathcal{A} to B and $\tilde{\exists}$ be a function from \mathcal{E} to B . The structure $\mathcal{B} = \langle B, B^+, \mathcal{A}, \mathcal{E}, \tilde{\top}, \tilde{\perp}, \Rightarrow, \tilde{\wedge}, \tilde{\vee}, \tilde{\forall}, \tilde{\exists} \rangle$ is said to be a truth values algebra if the set B^+ is closed by the intuitionistic deduction rules i.e. if for all a, b, c in B , A in \mathcal{A} and E in \mathcal{E} ,

1. if $a \Rightarrow b \in B^+$ and $a \in B^+$ then $b \in B^+$,
2. $a \Rightarrow b \Rightarrow a \in B^+$,
3. $(a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow a \Rightarrow c \in B^+$,
4. $\tilde{\top} \in B^+$,
5. $\tilde{\perp} \Rightarrow a \in B^+$,
6. $a \Rightarrow b \Rightarrow (a \tilde{\wedge} b) \in B^+$,
7. $(a \tilde{\wedge} b) \Rightarrow a \in B^+$,
8. $(a \tilde{\wedge} b) \Rightarrow b \in B^+$,
9. $a \Rightarrow (a \tilde{\vee} b) \in B^+$,
10. $b \Rightarrow (a \tilde{\vee} b) \in B^+$,
11. $(a \tilde{\vee} b) \Rightarrow (a \Rightarrow c) \Rightarrow (b \Rightarrow c) \Rightarrow c \in B^+$,
12. the set $a \Rightarrow A = \{a \Rightarrow e \mid e \in A\}$ is in \mathcal{A} and the set $E \Rightarrow a = \{e \Rightarrow a \mid e \in E\}$ is in \mathcal{A} ,
13. if all elements of A are in B^+ then $\tilde{\forall} A \in B^+$,
14. $\tilde{\forall} (a \Rightarrow A) \Rightarrow a \Rightarrow (\tilde{\forall} A) \in B^+$,
15. if $a \in A$, then $(\tilde{\forall} A) \Rightarrow a \in B^+$,
16. if $a \in E$, then $a \Rightarrow (\tilde{\exists} E) \in B^+$,
17. $(\tilde{\exists} E) \Rightarrow \tilde{\forall} (E \Rightarrow a) \Rightarrow a \in B^+$.

Proposition 1. Any Heyting algebra is a truth values algebra. The operations $\tilde{\top}$, $\tilde{\wedge}$, $\tilde{\vee}$ are greatest lower bounds, the operations $\tilde{\perp}$, $\tilde{\forall}$, $\tilde{\exists}$ are least upper bounds, the operation \Rightarrow is the arrow of the Heyting algebra, and $B^+ = \{\tilde{\top}\}$.

Proof. See [4].

Definition 3 (Full). A truth values algebra is said to be full if $\mathcal{A} = \mathcal{E} = \wp(B)$, i.e. if $\tilde{\forall} A$ and $\tilde{\exists} A$ exist for all subsets A of B .

Definition 4 (Ordered truth values algebra). An ordered truth values algebra is a truth values algebra together with a relation \sqsubseteq on B such that

- \sqsubseteq is an order relation, i.e. a reflexive, antisymmetric and transitive relation,
- B^+ is upward closed,
- $\tilde{\top}$ and $\tilde{\perp}$ are maximal and minimal elements,
- $\tilde{\wedge}$, $\tilde{\vee}$, $\tilde{\forall}$ and $\tilde{\exists}$ are monotone, \Rightarrow is left anti-monotone and right monotone.

Definition 5 (Complete ordered truth values algebra). A ordered truth values algebra is said to be complete if every subset of B has a greatest lower bound for \sqsubseteq .

2.3 Models

Definition 6 (\mathcal{B} -structure). Let $\mathcal{L} = \langle f_i, P_j \rangle$ be a language in predicate logic and \mathcal{B} be a truth values algebra, a \mathcal{B} -structure $\mathcal{M} = \langle M, \mathcal{B}, \hat{f}_i, \hat{P}_j \rangle$, for the language \mathcal{L} , is a structure such that \hat{f}_i is a function from M^n to M where n is the arity of the symbol f_i and \hat{P}_j is a function from M^n to \mathcal{B} where n is the arity of the symbol P_j .

This definition extends trivially to many-sorted languages.

Definition 7 (Denotation). Let \mathcal{B} be a truth values algebra, \mathcal{M} a \mathcal{B} -structure and ϕ an assignment. The denotation $\llbracket A \rrbracket_\phi$ of a proposition A in \mathcal{M} is defined as follows

- $\llbracket x \rrbracket_\phi = \phi(x)$,
- $\llbracket f(t_1, \dots, t_n) \rrbracket_\phi = \hat{f}(\llbracket t_1 \rrbracket_\phi, \dots, \llbracket t_n \rrbracket_\phi)$,
- $\llbracket P(t_1, \dots, t_n) \rrbracket_\phi = \hat{P}(\llbracket t_1 \rrbracket_\phi, \dots, \llbracket t_n \rrbracket_\phi)$,
- $\llbracket \top \rrbracket_\phi = \hat{\top}$,
- $\llbracket \perp \rrbracket_\phi = \hat{\perp}$,
- $\llbracket A \Rightarrow B \rrbracket_\phi = \llbracket A \rrbracket_\phi \hat{\Rightarrow} \llbracket B \rrbracket_\phi$,
- $\llbracket A \wedge B \rrbracket_\phi = \llbracket A \rrbracket_\phi \hat{\wedge} \llbracket B \rrbracket_\phi$,
- $\llbracket A \vee B \rrbracket_\phi = \llbracket A \rrbracket_\phi \hat{\vee} \llbracket B \rrbracket_\phi$,
- $\llbracket \forall x A \rrbracket_\phi = \hat{\forall} \{ \llbracket A \rrbracket_{\phi+(d/x)} \mid d \in M \}$,
- $\llbracket \exists x A \rrbracket_\phi = \hat{\exists} \{ \llbracket A \rrbracket_{\phi+(d/x)} \mid d \in M \}$.

Notice that the denotation of a proposition containing quantifiers may be undefined, but it is always defined if the truth values algebra is full.

Definition 8 (Denotation of a context and of a sequent). The denotation $\llbracket A_1, \dots, A_n \rrbracket_\phi$ of a context A_1, \dots, A_n is that of the proposition $A_1 \wedge \dots \wedge A_n$. The denotation $\llbracket A_1, \dots, A_n \vdash B \rrbracket_\phi$ of the sequent $A_1, \dots, A_n \vdash B$ is that of the proposition $(A_1 \wedge \dots \wedge A_n) \hat{\Rightarrow} B$.

Definition 9 (Model). A proposition A is said to be valid in a \mathcal{B} -structure \mathcal{M} , and the \mathcal{B} -structure \mathcal{M} is said to be a model of A if for all assignments ϕ , $\llbracket A \rrbracket_\phi$ is defined and is a positive truth value.

Consider a theory in deduction modulo defined by a set of axioms Γ and a congruence \equiv . The \mathcal{B} -structure \mathcal{M} is said to be a model of the theory Γ, \equiv if all axioms of Γ are valid in \mathcal{M} and for all terms or propositions A and B such that $A \equiv B$ and assignments ϕ , $\llbracket A \rrbracket_\phi$ and $\llbracket B \rrbracket_\phi$ are defined and $\llbracket A \rrbracket_\phi = \llbracket B \rrbracket_\phi$.

Deduction modulo is sound and complete with respect to this notion of model.

Proposition 2 (Soundness and completeness). The proposition A is provable in the theory formed with the axioms Γ and the congruence \equiv if and only if it is valid in all the models of Γ, \equiv where the truth values algebra is full, ordered and complete.

Proof. See [4].

2.4 Super-Consistency

Definition 10 (Super-consistent). *A theory in deduction modulo formed with the axioms Γ and the congruence \equiv is super-consistent if it has a \mathcal{B} -valued model for all full, ordered and complete truth values algebras \mathcal{B} .*

Proposition 3. *Simple type theory is super-consistent.*

Proof. Let \mathcal{B} be a full truth values algebra. We build the model \mathcal{M} as follows. The domain M_l is any non empty set, for instance the singleton $\{0\}$, the domain M_o is \mathcal{B} and the domain $M_{T \rightarrow U}$ is the set $M_U^{M_T}$ of functions from M_T to M_U . The interpretation of the symbols of the language is $\hat{S}_{T,U,V} = a \mapsto (b \mapsto (c \mapsto a(c)(b(c))))$, $\hat{K}_{T,U} = a \mapsto (b \mapsto a)$, $\hat{\alpha}(a, b) = a(b)$, $\hat{\varepsilon}(a) = a$, $\hat{\top} = \tilde{\top}$, $\hat{\perp} = \tilde{\perp}$, $\hat{\Rightarrow} = \tilde{\Rightarrow}$, $\hat{\wedge} = \tilde{\wedge}$, $\hat{\vee} = \tilde{\vee}$, $\hat{\forall}_T = a \mapsto \tilde{\forall}(Range(a))$, $\hat{\exists}_T = a \mapsto \tilde{\exists}(Range(a))$ where $Range(a)$ is the range of the function a . The model \mathcal{M} is a \mathcal{B} -valued model of simple type theory.

3 Cut Elimination

3.1 The Algebra of Sequents

Definition 11 (Neutral proof). *A proof is said to be neutral if its last rule is the axiom rule or an elimination rule, but not an introduction rule.*

Definition 12 (A positive definition of cut free proofs). *Cut free proofs are defined inductively as follows:*

- a proof that ends with the axiom rule is cut free,
- a proof that ends with an introduction rule and where the premises of the last rule are proved with cut free proofs is cut free,
- a proof that ends with an elimination rule and where the major premise of the last rule is proved with a neutral cut free proof and the other premises with cut free proofs is cut free.

Definition 13 (The algebra of sequents)

- $\tilde{\top}$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof or such that $C \equiv \top$.
- $\tilde{\perp}$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof.
- a $\tilde{\wedge} b$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof or such that $C \equiv (A \wedge B)$ with $(\Gamma \vdash A) \in a$ and $(\Gamma \vdash B) \in b$.
- a $\tilde{\vee} b$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof or such that $C \equiv (A \vee B)$ with $(\Gamma \vdash A) \in a$ or $(\Gamma \vdash B) \in b$.
- a $\tilde{\Rightarrow} b$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof or such that $C \equiv (A \Rightarrow B)$ and for all contexts Σ such that $(\Gamma, \Sigma \vdash A) \in a$, we have $(\Gamma, \Sigma \vdash B) \in b$.
- $\tilde{\forall} S$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof or such that $C \equiv (\forall x A)$ and for every term t and every a in S , $(\Gamma \vdash (t/x)A) \in a$.

- $\tilde{\exists} S$ is the set of sequents $\Gamma \vdash C$ that have a neutral cut free proof or such that $C \equiv (\exists x A)$ and for some term t and some a in S , $(\Gamma \vdash (t/x)A) \in a$.

Let S be the smallest set of sets of sequents closed by $\tilde{\top}$, $\tilde{\perp}$, $\tilde{\wedge}$, $\tilde{\vee}$, $\tilde{\Rightarrow}$, $\tilde{\forall}$, $\tilde{\exists}$ and by arbitrary intersections.

Proposition 4. *The structure $\mathcal{S} = \langle S, S, \wp(S), \wp(S), \tilde{\top}, \tilde{\perp}, \tilde{\Rightarrow}, \tilde{\wedge}, \tilde{\vee}, \tilde{\forall}, \tilde{\exists}, \subseteq \rangle$ is a full, ordered and complete truth values algebra.*

Proof. As all truth values are positive, the conditions of Definition 2 are obviously met. Thus \mathcal{S} is a truth values algebra. As the domains of $\tilde{\forall}$ and $\tilde{\exists}$ are defined as $\wp(S)$, this algebra is full. As it is closed by arbitrary intersections, all subsets of S have a greatest lower bound, thus all subsets of S have a least upper bound and the algebra is complete.

Remark. The algebra \mathcal{S} is not a Heyting algebra. In particular $\tilde{\top} \tilde{\wedge} \tilde{\top}$ and $\tilde{\top}$ are different: the first set contains the sequent $\vdash \top \wedge \top$, but not the second.

Proposition 5. *For all elements a of S , contexts Γ , and propositions A and B*

- $(\Gamma, A \vdash A) \in a$,
- if $(\Gamma \vdash B) \in a$ then $(\Gamma, A \vdash B) \in a$,
- if $(\Gamma \vdash A) \in a$ and $B \equiv A$ then $(\Gamma \vdash B) \in a$,
- if $(\Gamma \vdash A) \in a$ then $\Gamma \vdash A$ has a cut free proof.

Proof. The first proposition is proved by noticing that the sequent $\Gamma, A \vdash A$ has a neutral cut free proof. The others are proved by simple inductions on the construction of a .

Consider a super-consistent theory Γ, \equiv . As this theory is super-consistent, it has an \mathcal{S} -model \mathcal{M} . In the rest of the paper, \mathcal{M} refers to this fixed model whose domain is written M .

3.2 The Algebra of Contexts

Definition 14 (Fiber). *Let b be a set of sequents and A a proposition, we define the fiber over A in b , $b \triangleleft A$, as the set of contexts Γ such that $(\Gamma \vdash A) \in b$.*

Definition 15 (Outer value [10,11]). *Let A be a proposition, ϕ be an assignment and σ a substitution, we define the set of contexts $[A]_{\phi}^{\sigma}$ as the set $\llbracket A \rrbracket_{\phi} \triangleleft \sigma A$ i.e. $\{\Gamma \mid (\Gamma \vdash \sigma A) \in \llbracket A \rrbracket_{\phi}\}$.*

Proposition 6. *For all contexts Γ , propositions A and B , substitutions σ , and \mathcal{M} -assignments ϕ*

- $(\Gamma, \sigma A) \in [A]_{\phi}^{\sigma}$,
- if $\Gamma \in [B]_{\phi}^{\sigma}$ then $(\Gamma, A) \in [B]_{\phi}^{\sigma}$,
- if $\Gamma \in [A]_{\phi}^{\sigma}$ and $B \equiv A$ then $\Gamma \in [B]_{\phi}^{\sigma}$,
- if $\Gamma \in [A]_{\phi}^{\sigma}$ then $\Gamma \vdash \sigma A$ has a cut free proof.

Proof. From Proposition 5.

Definition 16 (The algebra of contexts). Let Ω be the smallest set of sets of contexts containing all the $[A]_{\phi}^{\sigma}$ for some proposition A , assignment ϕ , and substitution σ and closed by arbitrary intersections.

Notice that an element c of Ω can always be written in the form

$$c = \bigcap_{i \in \Lambda_c} [A_i]_{\phi_i}^{\sigma_i}$$

Proposition 7. The set Ω ordered by inclusion is a complete Heyting algebra.

Proof. As Ω is ordered by inclusion, the greatest lower bound of a subset of Ω is the intersection of all its elements. As Ω is closed by arbitrary intersections, all its subsets have greatest lower bounds. Thus, all its subsets also have least upper bounds.

The operations $\check{\top}$, $\check{\wedge}$ and $\check{\forall}$ are defined as nullary, binary and infinitary greatest lower bounds and the operations $\check{\perp}$, $\check{\vee}$ and $\check{\exists}$ are defined as nullary, binary and infinitary least upper bounds. Finally, the arrow \Rightarrow of two elements a and b is the least upper bound of all the c in Ω such that $a \cap c \leq b$

$$a \Rightarrow b = \check{\exists} \{c \in \Omega \mid a \cap c \leq b\}$$

Notice that the nullary least upper bound $\check{\perp}$ is the intersection of all the elements of Ω that contain the empty set, i.e. the intersection of all the elements of Ω .

The binary least upper bound, $a \check{\vee} b$, of a and b is the intersection of all the elements of Ω that contain $a \cup b$. From Definition 16

$$a \check{\vee} b = \bigcap_{(a \cup b) \subseteq c} c = \bigcap_{(a \cup b) \subseteq \bigcap [A_i]_{\phi_i}^{\sigma_i}} \left(\bigcap [A_i]_{\phi_i}^{\sigma_i} \right) = \bigcap_{(a \cup b) \subseteq [A]_{\phi}^{\sigma}} [A]_{\phi}^{\sigma}$$

The infinitary least upper bound $\check{\exists} E$ of the elements of a set E is the intersection of all the elements of Ω that contain the union of the elements of E . For the same reason as above

$$\check{\exists} E = \bigcap_{(\bigcup E) \subseteq c} c = \bigcap_{(\bigcup E) \subseteq [A]_{\phi}^{\sigma}} [A]_{\phi}^{\sigma}$$

Finally, notice that Ω is a non trivial Heyting algebra, although the Heyting algebra \mathcal{S}/S^+ is trivial because $S^+ = S$.

The next proposition, the Key lemma of our proof, shows that the outer values of compound propositions can be obtained from the outer values of their components using the suitable operation of the Heyting algebra Ω . Notice that, unlike most semantic cut elimination proofs, we directly prove equalities in this lemma, and not just inclusions, although the cut elimination proof is not completed yet.

Proposition 8 (Key lemma). For all substitutions σ , assignments ϕ and propositions A and B

$$- [\top]_{\phi}^{\sigma} = \check{\top},$$

- $[\perp]_\phi^\sigma = \check{\perp}$,
- $[A \wedge B]_\phi^\sigma = [A]_\phi^\sigma \check{\wedge} [B]_\phi^\sigma$,
- $[A \vee B]_\phi^\sigma = [A]_\phi^\sigma \check{\vee} [B]_\phi^\sigma$,
- $[A \Rightarrow B]_\phi^\sigma = [A]_\phi^\sigma \check{\Rightarrow} [B]_\phi^\sigma$,
- $[\forall x A]_\phi^\sigma = \check{\forall} \{ [A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M \}$,
- $[\exists x A]_\phi^\sigma = \check{\exists} \{ [A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M \}$.

where \mathcal{T} is the set of open terms in the language of this theory.

Proof. - By Definition [13](#), for any Γ , $(\Gamma \vdash \top) \in \check{\top}$. Thus $[\top]_\phi^\sigma = \Omega = \check{\top}$.

- The set $\check{\perp}$ is the intersection of all $[C]_\rho^\tau$. In particular, $\check{\perp} \subseteq [\perp]_\phi^\sigma$. Conversely, let $\Gamma \in [\perp]_\phi^\sigma$. Consider arbitrary C , ρ and τ . By Definition [13](#), $\Gamma \vdash \sigma \perp$ has a neutral cut free proof. So does $\Gamma \vdash \tau C$ and thus $\Gamma \in [C]_\rho^\tau$. Hence Γ is an element of all $[C]_\rho^\tau$ and therefore of their intersection $\check{\perp}$.
- Let $\Gamma \in [A]_\phi^\sigma \check{\wedge} [B]_\phi^\sigma = [A]_\phi^\sigma \cap [B]_\phi^\sigma$. We have $\Gamma \in [A]_\phi^\sigma$ and $\Gamma \in [B]_\phi^\sigma$ and thus $(\Gamma \vdash \sigma A) \in \llbracket A \rrbracket_\phi$ and $(\Gamma \vdash \sigma B) \in \llbracket B \rrbracket_\phi$. From Definition [13](#), we get $(\Gamma \vdash \sigma(A \wedge B)) \in \llbracket A \wedge B \rrbracket_\phi$. Hence $\Gamma \in [A \wedge B]_\phi^\sigma$. Conversely, let $\Gamma \in [A \wedge B]_\phi^\sigma$, we have $(\Gamma \vdash \sigma(A \wedge B)) \in (\llbracket A \rrbracket_\phi \check{\wedge} \llbracket B \rrbracket_\phi)$. If $\Gamma \vdash \sigma(A \wedge B)$ has a neutral and cut free proof, then so do $\Gamma \vdash \sigma A$ and $\Gamma \vdash \sigma B$. Thus $\Gamma \in [A]_\phi^\sigma$ and $\Gamma \in [B]_\phi^\sigma$, hence $\Gamma \in [A]_\phi^\sigma \cap [B]_\phi^\sigma = [A]_\phi^\sigma \check{\wedge} [B]_\phi^\sigma$. Otherwise we directly have $\Gamma \in [A]_\phi^\sigma$ and $\Gamma \in [B]_\phi^\sigma$ and we conclude the same way.
- Let us first prove $[A]_\phi^\sigma \check{\vee} [B]_\phi^\sigma \subseteq [A \vee B]_\phi^\sigma$. It is sufficient to prove that $[A \vee B]_\phi^\sigma$ is an upper bound of $[A]_\phi^\sigma$ and $[B]_\phi^\sigma$. Let $\Gamma \in [A]_\phi^\sigma$. We have $(\Gamma \vdash \sigma A) \in \llbracket A \rrbracket_\phi$. By Definition [13](#), $(\Gamma \vdash \sigma(A \vee B)) \in (\llbracket A \rrbracket_\phi \check{\vee} \llbracket B \rrbracket_\phi) = \llbracket A \vee B \rrbracket_\phi$. Thus $\Gamma \in [A \vee B]_\phi^\sigma$. We prove, in a similar way, that if $\Gamma \in [B]_\phi^\sigma$ then $\Gamma \in [A \vee B]_\phi^\sigma$. Conversely, let $\Gamma \in [A \vee B]_\phi^\sigma$. Let C , ρ and τ such that $[A]_\phi^\sigma \cup [B]_\phi^\sigma \subseteq [C]_\rho^\tau$. We have $(\Gamma \vdash \sigma(A \vee B)) \in (\llbracket A \rrbracket_\phi \check{\vee} \llbracket B \rrbracket_\phi)$. From Definition [13](#), there are three cases to consider. First, if $\Gamma \vdash \sigma(A \vee B)$ has a neutral cut free proof. As $(\Gamma, \sigma A) \in [A]_\phi^\sigma \subseteq [C]_\rho^\tau$, the sequent $\Gamma, \sigma A \vdash \tau C$ has a cut free proof by Proposition [6](#). In a similar way, the sequent $\Gamma, \sigma B \vdash \tau C$ has a cut free proof. Hence, we can apply the \vee -elim rule and obtain a neutral cut free proof of $\Gamma \vdash \tau C$. Thus $\Gamma \in [C]_\rho^\tau$. As Γ is an element of all such upper bounds, it is an element of their intersection i.e. of $[A]_\phi^\sigma \check{\vee} [B]_\phi^\sigma$. Second, if $(\Gamma \vdash \sigma A) \in \llbracket A \rrbracket_\phi$. We have $\Gamma \in [A]_\phi^\sigma \subseteq [C]_\rho^\tau$. Again, Γ is an element of their intersection. The case $(\Gamma \vdash \sigma B) \in \llbracket B \rrbracket_\phi$ is similar.
- Let us prove $[A \Rightarrow B]_\phi^\sigma \subseteq [A]_\phi^\sigma \check{\Rightarrow} [B]_\phi^\sigma$. This is equivalent to $[A]_\phi^\sigma \cap [A \Rightarrow B]_\phi^\sigma \subseteq [B]_\phi^\sigma$. Suppose $(\Gamma \vdash \sigma A) \in \llbracket A \rrbracket_\phi$ and $(\Gamma \vdash \sigma A \Rightarrow \sigma B) \in \llbracket A \Rightarrow B \rrbracket_\phi = \llbracket A \rrbracket_\phi \check{\Rightarrow} \llbracket B \rrbracket_\phi$. If $\Gamma \vdash \sigma A \Rightarrow \sigma B$ has a neutral cut free proof, so does $\Gamma \vdash \sigma B$, as $\Gamma \vdash \sigma A$ has a cut free proof. Thus $\Gamma \in [B]_\phi^\sigma$. Otherwise, considering an empty context Σ in Definition [13](#), we have $(\Gamma \vdash \sigma A) \in \llbracket A \rrbracket_\phi$ and thus we get $(\Gamma \vdash \sigma B) \in \llbracket B \rrbracket_\phi$. Conversely let us prove $[A]_\phi^\sigma \check{\Rightarrow} [B]_\phi^\sigma \subseteq [A \Rightarrow B]_\phi^\sigma$. We have to prove that $[A \Rightarrow B]_\phi^\sigma$ is an upper bound of the set of all the $c \in \Omega$ such that $c \cap [A]_\phi^\sigma \subseteq [B]_\phi^\sigma$. Let such a c , we have to prove $c \subseteq [A \Rightarrow B]_\phi^\sigma$. As noticed c has the form $\bigcap [C_i]_{\rho_i}^{\tau_i}$. Let $\Gamma \in c$. We must show $(\Gamma \vdash \sigma A \Rightarrow \sigma B) \in \llbracket A \Rightarrow B \rrbracket_\phi = \llbracket A \rrbracket_\phi \check{\Rightarrow} \llbracket B \rrbracket_\phi$. For this, let Σ such that $(\Gamma, \Sigma \vdash \sigma A) \in \llbracket A \rrbracket_\phi$. This

is equivalent to $\Gamma, \Sigma \in [A]_\phi^\sigma$. By Proposition 6, we know that $\Gamma, \Sigma \in [C_i]_{\rho_i}^{\tau_i}$. Therefore it is an element of their intersection. Thus $\Gamma, \Sigma \in [B]_\phi^\sigma$. Finally $(\Gamma, \Sigma \vdash \sigma B) \in \llbracket B \rrbracket_\phi^\sigma$. Hence $c \subseteq [A \Rightarrow B]_\phi^\sigma$.

- Let $\Gamma \in \bigcap \{[A]_{\phi+(d/x)}^{\sigma+(t/x)}, t \in \mathcal{T}, d \in M\}$. Then we have for any t and any d , $(\Gamma \vdash (\sigma + (t/x))A) \in \llbracket A \rrbracket_{\phi+(d/x)}$. Hence, $(\Gamma \vdash \sigma \forall x A) \in \tilde{\forall} \{ \llbracket A \rrbracket_{\phi+(d/x)}, d \in M \} = \llbracket \forall x A \rrbracket_\phi^\sigma$. Conversely, let $\Gamma \in \llbracket \forall x A \rrbracket_\phi^\sigma$. Then $(\Gamma \vdash \sigma \forall x A) \in \llbracket \forall x A \rrbracket_\phi^\sigma$. If $\Gamma \vdash \sigma \forall x A$ has a neutral cut free proof then so does the sequent $\Gamma \vdash (\sigma + (t/x))A$ for any t and this sequent is an element of $\llbracket A \rrbracket_{\phi+(d/x)}$ for any d . Hence $\Gamma \in [A]_{\phi+(d/x)}^{\sigma+(t/x)}$ for any t, d . So, it is an element of their intersection. Otherwise, by Definition 13, for all t and d we have $(\Gamma \vdash (\sigma + (t/x))A) \in \llbracket A \rrbracket_{\phi+(d/x)}$ thus Γ is an element of the intersection.
- We first prove that for any t, d , $[\exists x A]_\phi^\sigma$ is an upper bound of the set $\{[A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M\}$. Consider some t, d and $\Gamma \in [A]_{\phi+(d/x)}^{\sigma+(t/x)}$. We have $(\Gamma \vdash (\sigma + (t/x))A) \in \llbracket A \rrbracket_{\phi+(d/x)}$. By Definition 13, $(\Gamma \vdash \sigma \exists x A) \in \tilde{\exists} \{ \llbracket A \rrbracket_{\phi+(d/x)}, d \in M \}$. Hence $\Gamma \in [\exists x A]_\phi^\sigma$ and for any t and d , $[A]_{\phi+(d/x)}^{\sigma+(t/x)} \subseteq [\exists x A]_\phi^\sigma$. So $\tilde{\exists} \{ [A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M \} \subseteq [\exists x A]_\phi^\sigma$. Conversely, let $\Gamma \in [\exists x A]_\phi^\sigma$. Suppose $\Gamma \vdash \sigma \exists x A$ has a neutral cut free proof. Let $u = \bigcap [C_i]_{\rho_i}^{\tau_i}$ an upper bound of $\{[A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M\}$. We can choose $u = [C]_\rho^\tau$, since we need the intersection of the upper bounds. Let $\phi' = \phi + (d/x)$ and $\sigma' = \sigma + (y/x)$ where y is a variable appearing neither in ϕ nor in σ (the choice of $d \in M$ is immaterial). By hypothesis: $[A]_{\phi'}^{\sigma'} \subseteq [C]_\rho^\tau$. By Proposition 6, $\sigma' A \in [A]_{\phi'}^{\sigma'}$. Hence $\sigma' A \in [C]_\rho^\tau$. Thus the sequent $(\sigma' A \vdash \tau C) \in \llbracket C \rrbracket_\rho^\tau$ has a cut free proof by Proposition 6. Thus so does the sequent $\Gamma \vdash \tau C$. Hence $\Gamma \in [C]_\rho^\tau$. This is valid for any $[C]_\rho^\tau$ upper bound of $\{[A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M\}$. So, Γ is in their intersection, that is $\tilde{\exists} \{ [A]_{\phi+(d/x)}^{\sigma+(t/x)}, t \in \mathcal{T}, d \in M \}$ Otherwise by Definition 13, $\Gamma \vdash \sigma \exists x A$ is such that for some term t and element d , $(\Gamma \vdash (\sigma + (t/x))A) \in \llbracket A \rrbracket_{\phi+(d/x)}$. This shows that $\Gamma \in [A]_{\phi+(d/x)}^{\sigma+(t/x)}$. Then $\Gamma \in \tilde{\exists} \{ [A]_{\phi+(d/x)}^{\sigma+(t/x)} \mid t \in \mathcal{T}, d \in M \}$.

Proposition 9. $\sigma \Gamma \in [\Gamma]_\phi^\sigma$

Proof. Let $\Gamma = A_1, \dots, A_n$. Recall that, by Definition 8 and Proposition 8, $[\Gamma]_\phi^\sigma = [A_1 \wedge \dots \wedge A_n]_\phi^\sigma = [A_1]_\phi^\sigma \tilde{\wedge} \dots \tilde{\wedge} [A_n]_\phi^\sigma$. Using Proposition 6, we have $\sigma \Gamma \in [A_1]_\phi^\sigma, \dots, \sigma \Gamma \in [A_n]_\phi^\sigma$, thus $\sigma \Gamma \in ([A_1]_\phi^\sigma \tilde{\wedge} \dots \tilde{\wedge} [A_n]_\phi^\sigma)$.

3.3 Hybridization and Cut Elimination

A usual way to prove cut elimination would be to prove by induction over proof structure that if the sequent $\Gamma \vdash B$ is provable then for every substitution σ and every valuation ϕ , $[\Gamma]_\phi^\sigma \subseteq [B]_\phi^\sigma$. Then using the fact that $\Gamma \in [\Gamma]_\phi^\sigma$ (Proposition 9) we can prove $\Gamma \in [B]_\phi^\sigma$ and conclude, with Proposition 6, that $\Gamma \vdash B$ has a cut free proof.

We shall follow a slightly different way here and construct another model \mathcal{D} , built from \mathcal{M} and Ω -valued, such that the denotation of a proposition A is the value $[A]_{\phi}^{\sigma}$. Then the fact that if $\Gamma \vdash B$ is provable then $[\Gamma]_{\phi}^{\sigma} \subseteq [B]_{\phi}^{\sigma}$ will be just a consequence of the soundness theorem. The elements of the domain of the model \mathcal{D} are quite similar to the V-complexes used in the proofs of cut elimination for simple type theory that proceed by proving the completeness of the cut free calculus. So we give a generalization of the notion of V-complex that can be used not only for simple type theory but also for all super-consistent theories.

Consider a super-consistent theory and its model \mathcal{M} defined in Section 3.

Definition 17 (The model \mathcal{D}). *The model \mathcal{D} is an Ω -valued model with domain $D = \mathcal{T}' \times M$ where \mathcal{T}' is the set of (classes modulo \equiv of) open terms of the language of the theory and M the domain of the model \mathcal{M} .*

Let f be a function symbol of the language and $\hat{f}^{\mathcal{M}}$ its interpretation in the model \mathcal{M} , the interpretation $\hat{f}^{\mathcal{D}}$ of this symbol in the model \mathcal{D} is the function from D^n to D

$$\langle t_1, a_1 \rangle, \dots, \langle t_n, a_n \rangle \mapsto \langle f(t_1, \dots, t_n), \hat{f}^{\mathcal{M}}(a_1, \dots, a_n) \rangle$$

Let P be a predicate symbol of the language and $\hat{P}^{\mathcal{M}}$ its interpretation in the model \mathcal{M} . The interpretation $\hat{P}^{\mathcal{D}}$ of this symbol in the model \mathcal{D} is the function from D^n to Ω

$$\langle t_1, a_1 \rangle, \dots, \langle t_n, a_n \rangle \mapsto \hat{P}^{\mathcal{M}}(a_1, \dots, a_n) \triangleleft P(t_1, \dots, t_n)$$

Let ψ be an assignment mapping variables to elements $\langle t, d \rangle$ of D . We write ψ^1 for the substitution mapping the variable x to a fixed representative of the first component of ψx and ψ^2 for the \mathcal{M} -assignment mapping x to the second component of ψx . Notice that, by Proposition 6, $[A]_{\psi^2}^{\psi^1}$ is independent of the choice of the representatives in ψ^1 .

Proposition 10. *For any term t and assignment ψ*

$$\llbracket t \rrbracket_{\psi}^{\mathcal{D}} = \langle \psi^1 t, \llbracket t \rrbracket_{\psi^2}^{\mathcal{M}} \rangle$$

For any proposition A and assignment ψ

$$\llbracket A \rrbracket_{\psi}^{\mathcal{D}} = [A]_{\psi^2}^{\psi^1}$$

Proof. The first statement is proved by induction on the structure of t . The second by induction over the structure of A . If A is atomic, the result follows from the first statement, Definition 14 and Definition 17 and in all the other cases, from Proposition 8 and the induction hypothesis.

Proposition 11 (\mathcal{D} is a model of \equiv). *If $A \equiv B$, then $\llbracket A \rrbracket_{\psi}^{\mathcal{D}} = \llbracket B \rrbracket_{\psi}^{\mathcal{D}}$.*

Proof. From Proposition 6, we have $[A]_{\psi^2}^{\psi^1} = [B]_{\psi^2}^{\psi^1}$, and, by Proposition 10, we get $\llbracket A \rrbracket_{\psi}^{\mathcal{D}} = \llbracket B \rrbracket_{\psi}^{\mathcal{D}}$.

Proposition 12 (Completeness of the cut free calculus). *If the sequent $\Gamma \vdash B$ is valid in the model \mathcal{D} , then it has a cut free proof.*

Proof. Let $\Gamma = A_1, \dots, A_n$ and ϕ be an arbitrary \mathcal{M} -valuation. Let ψ be the \mathcal{D} -valuation mapping x to the pair $\langle x, \phi(x) \rangle$. If the sequent $\Gamma \vdash B$ is valid in \mathcal{D} , then the proposition $(A_1 \wedge \dots \wedge A_n) \Rightarrow B$ is valid in \mathcal{D} i.e. $\llbracket (A_1 \wedge \dots \wedge A_n) \Rightarrow B \rrbracket_{\psi}^{\mathcal{D}} = \checkmark$, i.e. $\llbracket A_1 \wedge \dots \wedge A_n \rrbracket_{\psi}^{\mathcal{D}} \subseteq \llbracket B \rrbracket_{\psi}^{\mathcal{D}}$. Using Proposition 10, $[\Gamma]_{\psi^1}^{\psi^1} \subseteq [B]_{\psi^2}^{\psi^1}$. Using Proposition 9, $\psi^1 \Gamma \in [B]_{\psi^2}^{\psi^1}$, i.e. $\Gamma \in [B]_{\psi^2}^{\psi^1}$. Using Proposition 6, $\Gamma \vdash B$ has a cut free proof.

Theorem 1 (Cut elimination). *If the sequent $\Gamma \vdash B$ is provable, then it has a cut free proof.*

Proof. From the soundness theorem (Proposition 2) and Proposition 1, if $\Gamma \vdash B$ is provable, then it is valid in all Heyting algebra-valued models of the congruence, and in particular in the model \mathcal{D} . Hence, by Proposition 12, it has a cut free proof.

4 Application to Simple Type Theory

As a particular case, we get a cut elimination proof for simple type theory.

Let us detail the model constructions in this case. Based on the language of simple type theory, we first build the truth values algebra of sequents \mathcal{S} of Definition 13. Then using the super-consistency of simple type theory, we build the model \mathcal{M} as in Proposition 3. In particular, we let $M_L = \{0\}$, $M_o = S$, and $M_{T \rightarrow U} = M_U^{M_T}$. Then, we let $D_T = \mathcal{T}'_T \times M_T$, where \mathcal{T}'_T is the set of classes of terms of sort T . In particular, we have $D_L = \mathcal{T}'_L \times \{0\}$ and $D_o = \mathcal{T}'_o \times S$.

This construction is reminiscent of the definition of V-complexes, that are also ordered pairs whose first component is a term. In particular, in the definition of V-complexes of [12][13][13][9], we also take $\mathcal{C}_L = \mathcal{T}'_L \times \{0\}$ but $\mathcal{C}_o = \mathcal{T}'_o \times \{\mathbf{false}, \mathbf{true}\}$. In the intuitionistic case [3], the set $\{\mathbf{false}, \mathbf{true}\}$ is replaced by a complete Heyting algebra.

The difference here is that instead of using the small algebra $\{\mathbf{false}, \mathbf{true}\}$ or a Heyting algebra, we use the larger truth values algebra \mathcal{S} of sequents.

Another difference is that, in our definition, we first define completely the hierarchy \mathcal{M} and then perform the hybridization with the terms. Terms and functions are more intricate in the definition of V-complexes as $\mathcal{C}_{T \rightarrow U}$ is defined as a set of pairs formed with a term t of type $T \rightarrow U$ and a function f from \mathcal{C}_T to \mathcal{C}_U such that $f\langle t', f' \rangle$ is a pair whose first component is tt' . In our definition, in contrast, $D_{T \rightarrow U}$ is the set of pairs formed with a term t of type $T \rightarrow U$ and an element of $M_{T \rightarrow U}$, i.e. a function from M_T to M_U , not from D_T to D_U . When we apply a pair $\langle t, f \rangle$ of $D_{T \rightarrow U}$ to a pair $\langle t', f' \rangle$ of D_T , we just apply component-wise t to t' and f to f' and get the pair $\langle tt', ff' \rangle$. With the usual V-complexes, the result of application is the pair $f\langle t', f' \rangle$ whose first component is indeed tt' , but whose second component depends on f, f' , and also t' . This

is indeed necessary since, in the algebra $\{\mathbf{false}, \mathbf{true}\}$ or in a Heyting algebra, \top and $\top \wedge \top$ have the same interpretation and thus in the usual V-complexes models, the interpretation of \top and $\top \wedge \top$ have the same second component. The only way to make the second component of $P(\top)$ and $P(\top \wedge \top)$ different (this is necessary, since they are not equivalent) is to make it depending on the first component. In our truth values algebra, in contrast, \top and $\top \wedge \top$ have different interpretations. Moreover, $\llbracket P(\top) \rrbracket^{\mathcal{D}} = \llbracket P(\top) \rrbracket^{\mathcal{M}} \triangleleft P(\top)$ whereas $\llbracket P(\top \wedge \top) \rrbracket^{\mathcal{D}} = \llbracket P(\top \wedge \top) \rrbracket^{\mathcal{M}} \triangleleft P(\top \wedge \top)$. This shows that $P(\top) \in \llbracket P(\top) \rrbracket^{\mathcal{D}}$, from Definition 13 since $P(\top) \vdash P(\top)$ has a neutral cut free proof. On the same way, $P(\top \wedge \top) \in \llbracket P(\top \wedge \top) \rrbracket^{\mathcal{D}}$ for the same reason. But one does have neither $P(\top \wedge \top) \in \llbracket P(\top) \rrbracket^{\mathcal{D}}$ nor $P(\top) \in \llbracket P(\top \wedge \top) \rrbracket^{\mathcal{D}}$. Hence these truth values are distinct and incomparable.

Thus the main difference between our hybrid model construction and that of the V-complexes is that we have broken this dependency of the right component of the pair obtained by applying $\langle t, f \rangle$ to $\langle t', f' \rangle$ with respect to t' leading to a simpler construction in two steps. The reason why we have been able to do so is that starting with a larger algebra for D_o , our semantic components are more informative and thus are sufficient to define the interpretation of larger terms.

5 Conclusion

In this paper, we have simplified the notion of V-complex introduced for proving cut elimination for simple type theory and shown that when we use truth values algebras, this notion boils down to hybridization. Once simplified this way, it can be generalized to all super-consistent theories. This allows to relate the normalization proofs using reducibility candidates to the semantic cut elimination proofs. The latter appear to be a simplification of the former, where each proof is replaced by its conclusion. In the cut elimination proof, however, hybridization has permitted to obtain the inclusion of the interpretation of Γ in that of B , when the sequent $\Gamma \vdash B$ is provable, as a corollary of the soundness theorem. It remains to understand if such a construction can also be carried out for the normalization proof.

References

1. Andrews, P.B.: Resolution in type theory. *The Journal of Symbolic Logic* 36(3), 414–432 (1971)
2. Church, A.: A formulation of the simple theory of types. *Journal of Symbolic Logic* 5, 56–68 (1940)
3. De Marco, M., Lipton, J.: Completeness and cut elimination in Church’s intuitionistic theory of types. *Journal of Logic and Computation* 15, 821–854 (2005)
4. Dowek, G.: Truth value algebras and proof normalization. In: *TYPES 2006*, Springer, Heidelberg (To appear 2006)
5. Dowek, G., Hardin, T., Kirchner, C.: Hol-lambda-sigma: an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science* 11, 1–25 (2001)

6. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. *Journal of Automated Reasoning* 31, 33–72 (2003)
7. Dowek, G., Werner, B.: Proof normalization modulo. *The Journal of Symbolic Logic* 68(4), 1289–1316 (2003)
8. Girard, J.-Y.: Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In: *Proceedings of the Second Scandinavian Logic Symposium (Univ. Oslo, Oslo, 1970)* vol. 63 of *Studies in Logic and the Foundations of Mathematics*, pp. 63–92. North-Holland, Amsterdam (1971)
9. Hermant, O.: Semantic cut elimination in the intuitionistic sequent calculus. In: *Urzyczyn, P. (ed.) TLCA 2005. LNCS*, vol. 3461, pp. 221–233. Springer, Heidelberg (2005)
10. Okada, M.: Phase semantic cut-elimination and normalization proofs of first- and higher-order linear logic. *Theoretical Computer Science* 227, 333–396 (1999)
11. Okada, M.: A uniform semantic proof for cut-elimination and completeness of various first and higher order logics. *Theoretical Computer Science* 281, 471–498 (2002)
12. Prawitz, D.: Hauptsatz for higher order logic. *The Journal of Symbolic Logic* 33, 452–457 (1968)
13. Takahashi, M.-o.: A proof of cut-elimination theorem in simple type-theory. *Journal of the Mathematical Society of Japan* 19(4), 399–410 (1967)

Bottom-Up Rewriting Is Inverse Recognizability Preserving

Irène Durand and Gérard Sénizergues

LaBRI, Université Bordeaux 1
351 Cours de la libération, 33405 Talence cedex, France

Abstract. For the whole class of linear term rewriting systems, we define *bottom-up rewriting* which is a restriction of the usual notion of rewriting. We show that bottom-up rewriting effectively inverse-preserves recognizability and analyze the complexity of the underlying construction. The *Bottom-Up* class (BU) is, by definition, the set of linear systems for which every derivation can be replaced by a bottom-up derivation. Membership to BU turns out to be undecidable; we are thus lead to define more restricted classes: the classes $SBU(k)$, $k \in \mathbb{N}$ of *Strongly Bottom-Up(k)* systems for which we show that membership is decidable. We define the class of *Strongly Bottom-Up* systems by $SBU = \bigcup_{k \in \mathbb{N}} SBU(k)$. We give a polynomial sufficient condition for a system to be in SBU. The class SBU contains (strictly) several classes of systems which were already known to inverse preserve recognizability.

1 Introduction

An important concept in term rewriting is the notion of *preservation of recognizability* through rewriting. Each identification of a more general class of systems preserving recognizability, yields almost directly a new decidable call-by-need class [9], decidability results for confluence, accessibility, joinability. Also, recently, this notion has been used to prove termination of systems for which none of the already known termination techniques work [13]. Such a preservation property is also a tool for studying the recognizable/rational subsets of various monoids which are defined by a presentation $\langle X, \mathcal{R} \rangle$, where X is a finite alphabet and \mathcal{R} a Thue system (see e.g. [17][18]).

Many such known classes have been defined by imposing *syntactical* restrictions on the rewrite rules. For instance, in *growing* systems [14][19] variables at depth strictly greater than 1 in the left-handside of a rule cannot appear in the corresponding right-handside. Finite-path Overlapping systems [25] are also defined by syntactic restrictions on the system. Previous works on semi-Thue systems proving a recognizability preservation property, were based on syntactic restrictions (see [2], [3], [1], [21], [23]).

Other works establish that some *strategies* i.e. restrictions on the derivations rather than on the rules, ensure preservation of recognizability. Various such strategies were studied in [11], [20], [24].

We rather follow here this second approach: we define a new rewriting strategy which we call *bottom-up rewriting* for linear term rewriting systems. The bottom-up derivations are, intuitively, those derivations in which the rules are applied from the bottom of

the term towards the top (this set of derivations contains strictly the bottom-up derivations of [20] and the one-pass leaf-started derivations of [11]). An important feature of this strategy, as opposed to the ones quoted above, is that it allows *overlaps* between successive applications of rules.

We define *bottom-up*(k) derivations for $k \in \mathbb{N}$ (bu(k) derivations for short) where a certain amount (limited by k) of top-down sequences of rules is allowed. Our main result is that bottom-up rewriting inverse-preserves recognizability (Theorem 2). We use a simulation argument which reduces our statement to the preservation of recognizability by ground systems (this preservation property is shown in [7]). Our proof is constructive *i.e.* gives an algorithm for computing an automaton recognizing the antecedents of a recognizable set of terms. We sketch an estimation of the complexity of the algorithm for $k = 1$.

We then define the class of Bottom-up systems (BU for short) consisting of the linear systems for which, there exists some fixed $k \geq 0$, such that every derivation between two terms can be replaced by a derivation which is bu(k). We show that membership to BU(k) is undecidable for $k \geq 1$ even for semi-Thue systems. We thus define the restricted class of *strongly bottom-up* systems for which we show decidable membership. We finally give a polynomial sufficient condition for a system to be in SBU.

Note that the class SBU is rather large since it contains strictly all the classes of semi-Thue systems quoted above (once translated as term rewriting systems where all symbols have arity 0 or 1), the linear growing systems of [14] and the class of LFPO⁻¹ (i.e. the linear systems which belong to the class FPO⁻¹ defined in [25]).

2 Preliminaries

Words. We denote by A^* the set of finite words over the alphabet A . The *empty* word is denoted by ε . A word u is a prefix of a word v iff there exists some $w \in A^*$ such that $v = uw$. We denote by $u \preceq v$ the fact that u is a prefix of v . We then note $v \setminus u := w$.

Terms. We assume the reader familiar with terms. We call *signature* a set of symbols with fixed arity. For every $m \in \mathbb{N}$, \mathcal{F}_m denotes the subset of symbols of arity m . As usual, a *tree-domain* is a subset of \mathbb{N}^* , which is downwards closed for prefix ordering and left-brother ordering. Let us call $P' \subseteq P$ a *subdomain* of P iff, P' is a domain and, for every $u \in P, i \in \mathbb{N} (u \cdot i \in P' \ \& \ u \cdot (i + 1) \in P) \Rightarrow u \cdot (i + 1) \in P'$. A *term* on a signature \mathcal{F} is a partial map $t : \mathbb{N}^* \rightarrow \mathcal{F}$ whose domain is a tree-domain and which respects the arities. The domain of t is also called its set of *positions* and denoted by $\mathcal{Pos}(t)$. We write $\mathcal{Pos}^+(t)$ for $\mathcal{Pos}(t) \setminus \{\varepsilon\}$. If $u, v \in \mathcal{Pos}(t)$ and $u \preceq v$, we say that u is an *ancestor* of v in t . Given $v \in \mathcal{Pos}^+(t)$, its *father* is the position u such that $v = uw$ and $|w| = 1$. Given a term t and $u \in \mathcal{Pos}(t)$ the *subterm of t at u* is denoted by t/u and defined by $\mathcal{Pos}(t/u) = \{w \mid uw \in \mathcal{Pos}(t)\}$ and $\forall w \in \mathcal{Pos}(t/u), t/u(w) = t(uw)$. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of first-order terms built upon a signature \mathcal{F} and a denumerable set of variables \mathcal{V} . The set of variables of a term t is denoted by $\mathcal{Var}(t)$. The set of variable positions (resp. non variable positions) of a term t is denoted by $\mathcal{Pos}_{\mathcal{V}}(t)$ (resp. $\mathcal{Pos}_{\overline{\mathcal{V}}}(t)$). The *depth* of a term t is defined by: $dpt(t) := \sup\{|u| \mid u \in \mathcal{Pos}_{\overline{\mathcal{V}}}(t)\}$. We denote by $|t| := \text{Card}(\mathcal{Pos}(t))$ the *size* of a term t . A term which does not contain twice the same variable is called *linear*. Given a

linear term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $x \in \mathcal{V}\text{ar}(t)$, we shall denote by $\text{pos}(t, x)$ the position of x in t . A term containing no variable is called *ground*. The set of ground terms is abbreviated to $\mathcal{T}(\mathcal{F})$ or \mathcal{T} whenever \mathcal{F} is understood. Among all the variables, there is a special one designated by \square . A term containing exactly one occurrence of \square is called a *context*. A context is usually denoted as $C[\]$. If u is the position of \square in $C[\]$, $C[t]$ denotes the term $C[\]$ where t has been substituted at position u .

Term rewriting. A rewrite rule is a pair $l \rightarrow r$ of terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ which satisfies $\mathcal{V}\text{ar}(r) \subseteq \mathcal{V}\text{ar}(l)$. We call l (resp. r) the *left-handside* (resp. *right-handside*) of the rule (*lhs* and *rhs* for short). A rule is *linear* if both its left and right-handside are linear. A rule is *left-linear* if its left-handside is linear.

A *term rewriting system* (system for short) is a pair $(\mathcal{R}, \mathcal{F})$ where \mathcal{F} is a signature and \mathcal{R} a set of rewrite rules built upon the signature \mathcal{F} . When \mathcal{F} is clear from the context or contains exactly the symbols in \mathcal{R} , we may omit \mathcal{F} and write simply \mathcal{R} . We call *size* of the set of rules \mathcal{R} the number $\|\mathcal{R}\| := \sum_{l \rightarrow r \in \mathcal{R}} |l| + |r|$. Rewriting is defined as usual. A system is *linear* (resp. *left-linear*) if each of its rules is linear (resp. left-linear). A system \mathcal{R} is *growing* [14] if every variable of a right-handside is at depth at most 1 in the corresponding left-handside.

Automata. We shall consider exclusively bottom-up term (tree) automata [4] (which we abbreviate to *f.t.a*). An automaton \mathcal{A} is given by a 4-tuple $(\mathcal{F}, Q, Q_f, \Gamma)$ where \mathcal{F} is the signature, Q the set of states, Q_f the set of final states, Γ the set of transitions. The *size* of \mathcal{A} is defined by: $\|\mathcal{A}\| := \text{Card}(\Gamma) + \text{Card}(Q)$. The set of rules Γ can be viewed as a rewriting system over the signature $\mathcal{F} \cup Q$. We then denote by \rightarrow_Γ or by $\rightarrow_{\mathcal{A}}$ (resp. by \rightarrow_Γ^* or by $\rightarrow_{\mathcal{A}}^*$) the one-step rewriting relation (resp. the rewriting relation) generated by Γ . Given an automaton \mathcal{A} , let $L(\mathcal{A})$ be the set of terms accepted by \mathcal{A} . A set of terms T is *recognizable* if there exists a term automaton \mathcal{A} such that $T = L(\mathcal{A})$. The following technical normal form for *f.t.a* will be useful in our proofs.

Definition 1. A *n.f.t.a* $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Gamma)$ is called *standard* iff $\mathcal{F}_0 \subseteq Q$ and

- 1- Every rule of \mathcal{A} has the form $f(q_1, \dots, q_m) \rightarrow q$ with $m \geq 1$, $f \in \mathcal{F}_m$, $q_i, q \in Q$
- 2- For every $m \geq 1$, $f \in \mathcal{F}_m$, $q_1, \dots, q_m \in Q$ there exists a unique $q \in Q$ such that $f(q_1, \dots, q_m) \rightarrow_{\mathcal{A}} q$.

Automata and rewriting Given a system \mathcal{R} and a set of terms T , we define

$$\begin{aligned} (\rightarrow_{\mathcal{R}}^*)[T] &= \{s \in \mathcal{T}(\mathcal{F}) \mid \exists t \in T, s \rightarrow_{\mathcal{R}}^* t\} \text{ and} \\ [T](\rightarrow_{\mathcal{R}}^*) &= \{s \in \mathcal{T}(\mathcal{F}) \mid \exists t \in T, t \rightarrow_{\mathcal{R}}^* s\}. \end{aligned}$$

A system \mathcal{R} is *recognizability preserving* (resp. *inverse recognizability preserving*) if $[T](\rightarrow_{\mathcal{R}}^*)$ (resp. $(\rightarrow_{\mathcal{R}}^*)[T]$) is recognizable for every recognizable T .

Lemma 1. Let \mathcal{A} be a standard *f.t.a* over the signature \mathcal{F} . Let $t, t_1, t_2 \in \mathcal{T}(\mathcal{F} \cup Q)$. If $t \rightarrow_{\mathcal{A}}^* t_1, t \rightarrow_{\mathcal{A}}^* t_2$ and $\text{Pos}(t_1) = \text{Pos}(t_2)$, then $t_1 = t_2$.

Definition 2 (\mathcal{A} -reduct). Let \mathcal{A} be a standard *f.t.a* over the signature \mathcal{F} . Let $t \in \mathcal{T}(\mathcal{F} \cup Q)$ and let P be a subdomain of $\text{Pos}(t)$. We define $\text{Red}(t, P) = t'$ as the unique element of $\mathcal{T}(\mathcal{F} \cup Q)$ such that

$$\begin{aligned} 1 - t &\rightarrow_{\mathcal{A}}^* t' \\ 2 - \mathcal{P}\text{os}(t') &= P \end{aligned}$$

Lemma 2. *Let \mathcal{A} be a standard f.t.a over the signature \mathcal{F} . Let $t, t_1, t_2 \in \mathcal{T}(\mathcal{F} \cup Q)$. If $t \rightarrow_{\mathcal{A}}^* t_1, t \rightarrow_{\mathcal{A}}^* t_2$ and $\mathcal{P}\text{os}(t_1) \subseteq \mathcal{P}\text{os}(t_2)$, then $t_2 \rightarrow_{\mathcal{A}}^* t_1$.*

3 Bottom-Up Rewriting

In order to define *bottom-up rewriting*, we need some marking tools. In the following we assume that \mathcal{F} is a signature. We shall illustrate many of our definitions with the following system

$$\text{Example 1. } \mathcal{R}_1 = \{f(x) \rightarrow g(x), g(h(x)) \rightarrow i(x), i(x) \rightarrow a\}$$

3.1 Marking

We mark the symbols of a term using natural integers (as done in [12] for example).

Definition 3. *We define the (infinite) signature of marked symbols: $\mathcal{F}^{\mathbb{N}} := \{f^i \mid f \in \mathcal{F}, i \in \mathbb{N}\}$.*

For every integer $k \geq 0$ we note: $\mathcal{F}^{\leq k} := \{f^i \mid f \in \mathcal{F}, 0 \leq i \leq k\}$. The operation $m()$ returns the mark of a marked symbol: $m(f^i) = i$.

Definition 4. *The terms in $\mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ are called marked terms.*

The operation $m()$ extends to marked terms: if $t \in \mathcal{V}$, $m(t) = 0$, otherwise, $m(t) = m(t(\varepsilon))$. For every $f \in \mathcal{F}$, we identify f^0 and f ; it follows that $\mathcal{F} \subset \mathcal{F}^{\mathbb{N}}$, $\mathcal{T}(\mathcal{F}) \subset \mathcal{T}(\mathcal{F}^{\mathbb{N}})$ and $\mathcal{T}(\mathcal{F}, \mathcal{V}) \subset \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$.

$$\text{Example 2. } m(a^2) = 2, m(i(a^2)) = 0, m(h^1(a)) = 1, m(h^1(x)) = 1, m(x) = 0.$$

Definition 5. *Given $t \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ and $i \in \mathbb{N}$, we define the marked term t^i whose marks are all equal to i (except at variables) by:*

$$\mathcal{P}\text{os}(t^i) := \mathcal{P}\text{os}(t), \quad t^i(u) = t(u)^i \text{ if } t(u) \notin \mathcal{V}, \quad t^i(u) = t(u) \text{ if } t(u) \in \mathcal{V}.$$

This marking extends to sets of terms S ($S^i := \{t^i \mid t \in S\}$) and substitutions σ ($\sigma^i : x \mapsto (x\sigma)^i$). We use the notation: $m\max(t) := \max\{m(t/u) \mid u \in \mathcal{P}\text{os}(t)\}$. In the sequel, given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, \bar{t} will always refer to a term of $\mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ such that $\bar{t}^0 = t$. The same rule will apply to substitutions and contexts. Note that there are several possible \bar{t} for a single t .

Finite automata and marked terms. Given a finite tree-automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Gamma)$ we extend it over the signature $\mathcal{F}^{\mathbb{N}}$, by setting

$$\Gamma^{\mathbb{N}} := \{(f^j(q_1^{j_1}, \dots, q_m^{j_m}) \rightarrow q^j) \mid (f(q_1, \dots, q_m) \rightarrow q) \in \Gamma, j, j_1, \dots, j_m \in \mathbb{N}\},$$

and $\mathcal{A}^{\mathbb{N}} := (\mathcal{F}^{\mathbb{N}}, Q^{\mathbb{N}}, Q_f^{\mathbb{N}}, \Gamma^{\mathbb{N}})$. The binary relation $\rightarrow_{\mathcal{A}^{\mathbb{N}}}$ is an extension of $\rightarrow_{\mathcal{A}}$ to $\mathcal{T}(\mathcal{F}^{\mathbb{N}}) \times \mathcal{T}(\mathcal{F}^{\mathbb{N}})$. We will often note simply $\rightarrow_{\mathcal{A}}$ what should be denoted $\rightarrow_{\mathcal{A}^{\mathbb{N}}}$.

\mathbb{N} acts on marked terms. We define a right-action \odot of the monoid $(\mathbb{N}, \max, 0)$ over the set $\mathcal{F}^{\mathbb{N}}$ which just consists in applying the operation \max on every mark $i.e$ for every $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$, $n \in \mathbb{N}$,

$$\mathcal{Pos}(\bar{t} \odot n) := \mathcal{Pos}(\bar{t}), \quad \forall u \in \mathcal{Pos}(\bar{t}), \mathfrak{m}((\bar{t} \odot n)/u) := \max(\mathfrak{m}(\bar{t}/u), n), \quad (\bar{t} \odot n)^0 = \bar{t}^0$$

Lemma 3. *Let \mathcal{A} be some finite tree automaton over \mathcal{F} , $\bar{s}, \bar{t} \in \mathcal{T}((\mathcal{F} \cup Q)^{\mathbb{N}})$ and $n \in \mathbb{N}$. If $\bar{s} \rightarrow_{\mathcal{A}}^* \bar{t}$ then $(\bar{s} \odot n) \rightarrow_{\mathcal{A}}^* (\bar{t} \odot n)$.*

Marked rewriting. We define here the rewrite relation $\circ \rightarrow$ between marked terms. Figure 1 illustrates this definition.

For every linear marked term $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ and variable $x \in \mathcal{V}\text{ar}(\bar{t})$, we define:

$$M(\bar{t}, x) := \sup\{\mathfrak{m}(\bar{t}/w) \mid w < \text{pos}(\bar{t}, x)\} + 1. \quad (1)$$

Let \mathcal{R} be a left-linear system, $\bar{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$. Let us suppose that $\bar{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$ decomposes as

$$\bar{s} = \overline{C}[\bar{l}\bar{\sigma}]_v, \quad \text{with } (l, r) \in \mathcal{R}, \quad (2)$$

for some marked context $\overline{C}[\]_v$ and substitution $\bar{\sigma}$. We define a new marked substitution $\overline{\overline{\sigma}}$ (such that $\overline{\overline{\sigma}}^0 = \bar{\sigma}^0$) by: for every $x \in \mathcal{V}\text{ar}(r)$,

$$x\overline{\overline{\sigma}} := (x\bar{\sigma}) \odot M(\overline{C}[\bar{l}], x). \quad (3)$$

We then write $\bar{s} \circ \rightarrow \bar{t}$ where

$$\bar{s} = \overline{C}[\bar{l}\bar{\sigma}], \quad \bar{t} = \overline{C}[r\overline{\overline{\sigma}}]. \quad (4)$$

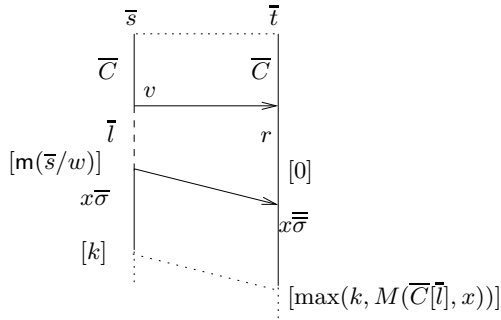


Fig. 1. A marked rewriting step

More precisely, an ordered pair of marked terms (\bar{s}, \bar{t}) is linked by the relation $\circ \rightarrow$ iff, there exists $\overline{C}[\]_v$, (l, r) , \bar{l} , $\bar{\sigma}$ and $\overline{\overline{\sigma}}$ fulfilling equations (2,4). A mark k will roughly mean that there were k successive applications of rules, each one with a leaf of the left-handside at a position strictly greater than a leaf of the previous right-handside.

The map $\bar{s} \mapsto \bar{s}^0$ (from marked terms to unmarked terms) extends into a map from marked derivations to unmarked derivations: every

$$\bar{s}_0 = \bar{C}_0[\bar{l}_0\bar{\sigma}_0]_{v_0} \circ \rightarrow \bar{C}_0[r_0\bar{\sigma}_0]_{v_0} = \bar{s}_1 \circ \rightarrow \dots \circ \rightarrow \bar{C}_{n-1}[r_{n-1}\bar{\sigma}_{n-1}]_{v_{n-1}} = \bar{s}_n \quad (5)$$

is mapped to the derivation

$$s_0 = C_0[l_0\sigma_0]_{v_0} \rightarrow C_0[r_0\sigma_0]_{v_0} = s_1 \rightarrow \dots \rightarrow C_{n-1}[r_{n-1}\sigma_{n-1}]_{v_{n-1}} = s_n. \quad (6)$$

The context \bar{C}_i , the rule (l_i, r_i) and the substitution $\bar{\sigma}_i$ completely determines \bar{s}_{i+1} . Thus, for every fixed pair (\bar{s}_0, s_0) , this map is a bijection from the set of derivations (5) starting from \bar{s}_0 , to the set of derivations (6) starting from s_0 .

Example 3. With the system \mathcal{R}_1 of Example 1 we get the following marked derivation:

$$f(h(f(h(a)))) \circ \rightarrow f(h(g(h^1(a^1)))) \circ \rightarrow f(h(i(a^2))) \circ \rightarrow f(h(a)) \circ \rightarrow g(h^1(a^1)) \circ \rightarrow i(a^2) \circ \rightarrow a$$

From now on, each time we deal with a derivation $s \rightarrow^* t$ between two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we may implicitly decompose it as (6) where n is the length of the derivation, $s = s_0$ and $t = s_n$.

3.2 Bottom-Up Derivations

Definition 6. The marked derivation (5) is weakly bottom-up if, for every $0 \leq j < n$, $l_j \notin \mathcal{V} \Rightarrow m(\bar{l}_j) = 0$, and $l_j \in \mathcal{V} \Rightarrow \sup\{m(\bar{s}_j/u) \mid u < v_j\} = 0$.

Definition 7. The derivation (6) is weakly bottom-up if the corresponding marked derivation (5) starting on the same term $\bar{s} = s$ is weakly bottom-up (following the above definition).

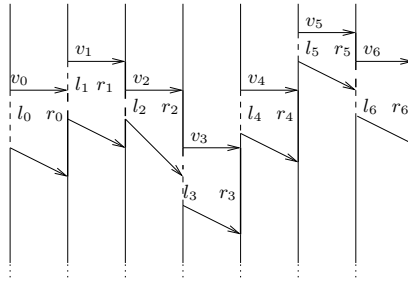


Fig. 2. A wbu-derivation

We shall abbreviate “weakly bottom-up” to wbu.

Lemma 4. Let \mathcal{R} be a linear system. If $s \rightarrow^*_{\mathcal{R}} t$ then there exists a wbu-derivation between s and t .

The linear restriction cannot be relaxed: let $\mathcal{R} = \{f(x) \rightarrow g(x, x), a \rightarrow b\}$; we have $f(a) \rightarrow g(a, a) \rightarrow g(b, a)$ but no wbu-derivation between $f(a)$ and $g(b, a)$.

Definition 8. A marked term \bar{s} is called *m-increasing* iff, for every $u, v \in \mathcal{P}\text{os}(\bar{s})$, $u \preceq v \Rightarrow m(\bar{s}/u) \leq m(\bar{s}/v)$.

Intuitively, in an *m-increasing* term, deeper positions carry the same or higher marks.

Lemma 5. Let $\bar{s} \circ \rightarrow^* \bar{t}$ be a marked wbu-derivation between $\bar{s}, \bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$. If \bar{s} is *m-increasing*, then \bar{t} is *m-increasing* too.

We classify the derivations according to the maximal value of the marks. We abbreviate “bottom-up” as bu.

Definition 9. A derivation is $\text{bu}(k)$ (resp. $\text{bu}^-(k)$) if it is wbu and, in the corresponding marked derivation $\forall i, 0 \leq i \leq n$, $\text{mmax}(\bar{s}_i) \leq k$ (resp. $\forall i, 0 \leq i < n$, $\text{mmax}(\bar{l}_i) < k$).

Notation $\bar{s} \circ \rightarrow^*_{\mathcal{R}} \bar{t}$ means that there exists a wbu marked derivation from s to t where all the marks belong to $[0, k]$.

Notation $s \rightarrow^*_k t$ means that there exists a $\text{bu}(k)$ derivation from s to t .

Example 4. For the system $\mathcal{R}_0 = \{f(f(x)) \rightarrow f(x)\}$ with the signature $\mathcal{F} = \{a^{(0)}, f^{(1)}\}$, although we may get a $\text{bu}(k)$ derivation for a term of the form $f(\dots f(a)\dots)$ with $k+1$ f symbols: $f(f(f(f(a)))) \circ \rightarrow f(f^1(f^1(a^1))) \circ \rightarrow f(f^2(a^2)) \circ \rightarrow f(a^3)$ we can always achieve a $\text{bu}^-(1)$ derivation: $f(f(f(f(a)))) \circ \rightarrow f(f(f(a^1))) \circ \rightarrow f(f(a^1)) \circ \rightarrow f(a^1)$

3.3 Bottom-Up Systems

We introduce here a hierarchy of classes of rewriting systems, based on their ability to meet the bottom-up restriction over derivations.

Definition 10. Let P be some property of derivations. A system $(\mathcal{R}, \mathcal{F})$ is called P if $\forall s, t \in \mathcal{T}(\mathcal{F})$ such that $s \rightarrow^*_t t$ there exists a P -derivation from s to t .

We denote by $\text{BU}(k)$ the class of $\text{BU}(k)$ systems, by $\text{BU}^-(k)$ the class of $\text{BU}^-(k)$ systems. One can check that, for every $k > 0$, $\text{BU}(k-1) \subsetneq \text{BU}^-(k) \subsetneq \text{BU}(k)$. Finally, the class of *bottom-up systems*, denoted BU , is defined by: $\text{BU} = \bigcup_{k \in \mathbb{N}} \text{BU}(k)$.

Example 5. The system $\mathcal{R}_0 = \{f(f(x)) \rightarrow f(x)\} \in \text{BU}^-(1)$ and \mathcal{R}_0 is not growing. The system \mathcal{R}_1 of Example 1 belongs to $\text{BU}^-(2)$ and \mathcal{R}_1 is not growing. The system $\mathcal{R}_2 = \{f(x) \rightarrow g(x), h(g(a)) \rightarrow a\}$ is growing and belongs to $\text{BU}^-(1)$. The system $\mathcal{R}_3 = \{f(x) \rightarrow g(x), g(h(x)) \rightarrow a\}$ is growing and belongs to $\text{BU}(1)$.

4 Bottom-Up Rewriting Is Inverse Recognizability Preserving

Let us recall the following classical result about ground rewriting systems

Theorem 1 ([7][6]). *Every ground system is inverse-recognizability preserving.*

The main theorem of this section (and of the paper) is the following extension of Theorem 1 to $\text{bu}(k)$ derivations of linear rewriting systems

Theorem 2. *Let \mathcal{R} be some linear rewriting system over the signature \mathcal{F} , let T be some recognizable subset of $\mathcal{T}(\mathcal{F})$ and let $k \geq 0$. Then, the set $(\rightarrow^*_k)_{\mathcal{R}}[T]$ is recognizable too.*

4.1 Construction

In order to prove Theorem [2](#) we are to introduce some technical definitions, and to prove some technical lemmas. Let us fix, from now on and until the end of the subsection, a linear system $(\mathcal{R}, \mathcal{F})$, a language $T \subseteq \mathcal{T}(\mathcal{F})$ recognized by a finite automaton over the extended signature $\mathcal{F} \cup \{\square\}$, $\mathcal{A} = (\mathcal{F} \cup \{\square\}, Q, Q_f, \Gamma)$ and an integer $k \geq 0$. In order to make the proofs easier, we assume in the first step of this subsection that:

$$\forall l \rightarrow r \in \mathcal{R}, l \notin \mathcal{V}, \text{ and } \mathcal{A} \text{ is standard.} \tag{7}$$

Let us define the integer $d := \max\{dpt(l) \mid l \rightarrow r \in \mathcal{R}\}$. We introduce a ground system \mathcal{S} which will be enough, together with \mathcal{A} , for describing the set of terms which rewrite in $L(\mathcal{A})$.

Definition 11. We define \mathcal{S} as the ground rewriting system over $\mathcal{T}((\mathcal{F} \cup Q)^{\leq k})$ consisting of all the rules of the form: $\bar{l}\bar{\tau} \rightarrow r\bar{\tau}$ where $l \rightarrow r$ is a rule of \mathcal{R}

$$m(\bar{l}) = 0 \tag{8}$$

and $\bar{\tau}, \bar{\tau}' : \mathcal{V} \rightarrow \mathcal{T}((\mathcal{F} \cup Q)^{\leq k})$ are marked substitutions such that, $\forall x \in \mathcal{V}ar(l)$

$$x\bar{\tau}' = (x\bar{\tau}) \odot M(\bar{l}, x), \quad dpt(x\bar{\tau}') \leq k \cdot d. \tag{9}$$

Lemma 6 (lifting $\mathcal{S} \cup \mathcal{A}$ to \mathcal{R}).

Let $\bar{s}, \bar{s}', \bar{t} \in \mathcal{T}((\mathcal{F} \cup Q)^{\leq k})$.

If $\bar{s}' \xrightarrow{*}_{\mathcal{A}} \bar{s}$ and $\bar{s} \xrightarrow{*}_{\mathcal{S} \cup \mathcal{A}} \bar{t}$ then, there exists a term $\bar{t}' \in \mathcal{T}((\mathcal{F} \cup Q)^{\leq k})$ such that $\bar{s}' \xrightarrow{k \circ}_{\mathcal{R}} \bar{t}'$ and $\bar{t}' \xrightarrow{*}_{\mathcal{A}} \bar{t}$.

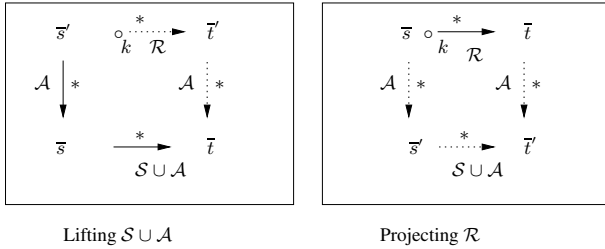


Fig. 3. Lemma [6](#) and [8](#)

Proof. 1- Let us prove that the lemma holds for $\bar{s} \xrightarrow{*}_{\mathcal{S} \cup \mathcal{A}} \bar{t}$. Under the assumption that $\bar{s}' \xrightarrow{*}_{\mathcal{A}} \bar{s}$ and $\bar{s} \xrightarrow{*}_{\mathcal{A}} \bar{t}$, we can just choose $\bar{t}' = \bar{s}'$ and obtain the conclusion.

Suppose now that $\bar{s} \xrightarrow{*}_{\mathcal{S}} \bar{t}$. This means that

$$\bar{s} = \bar{C}[\bar{l}\bar{\tau}], \quad \bar{t} = \bar{C}[r\bar{\tau}]$$

for some rule $l \rightarrow r \in \mathcal{R}$, marked context \bar{C} , and marked substitutions $\bar{\tau}, \bar{\tau}'$, satisfying [\(8-9\)](#). Since $\bar{s}' \xrightarrow{*}_{\mathcal{A}} \bar{s}$, we must have $\bar{s}' = \bar{C}[\bar{l}\bar{\tau}']$ where, for every $x \in \mathcal{V}ar(l)$, $x\bar{\tau}' \xrightarrow{*}_{\mathcal{A}} x\bar{\tau}$. Let us set

$$x\bar{\tau}' := x\bar{\tau}' \odot M(\bar{l}, x), \quad \bar{t}' := \bar{C}[r\bar{\tau}'].$$

The relation $\bar{s}' \circ \rightarrow_{\mathcal{R}} \bar{t}'$ holds (by definition of $\circ \rightarrow$) and this one-step derivation is wbu (by condition (8)). Moreover, since every mark of \bar{t} is $\leq k$ and every mark of \bar{s}' is $\leq k$, we are sure that, for every $x \in \mathcal{V}\text{ar}(l) \cap \mathcal{V}\text{ar}(r)$, $M(\bar{l}, x) \leq k$ (because some non smaller mark occurs in \bar{t}) and $\text{mmax}(x\bar{r}') \leq k$ (because this mark occurs in \bar{s}'), hence $\text{mmax}(x\bar{r}') \leq k$ which ensures that $\text{mmax}(\bar{t}') \leq k$ and finally:

$$\bar{s}' \stackrel{k}{\circ} \rightarrow_{\mathcal{R}} \bar{t}'.$$

By Lemma 3 for every $x \in \mathcal{V}\text{ar}(l)$, $x\bar{r}' = x\bar{r}' \odot M(\bar{l}, x) \rightarrow_{\mathcal{A}}^* x\bar{r} \odot M(\bar{l}, x) = x\bar{r}$. Hence $\bar{t}' = \overline{C[r\bar{r}']} \rightarrow_{\mathcal{A}}^* \overline{C[r\bar{r}]} = \bar{t}$.

2- The lemma can be deduced from point 1 above by induction on the integer n such that $\bar{s} \rightarrow_{S \cup \mathcal{A}}^n \bar{t}$.

Definition 12 (Top domain of a term). Let $\bar{t} \in \mathcal{T}((\mathcal{F} \cup Q)^{\leq k}, \{\square\})$. We define the top domain of \bar{t} , denoted by $\text{Topd}(\bar{t})$ as: $u \in \text{Topd}(\bar{t})$ iff

1- $u \in \text{Pos}(\bar{t})$

2- $\forall u_1, u_2 \in \mathbb{N}^*$ such that $u = u_1 \cdot u_2$, either $\text{m}(\bar{t}/u_1) = 0$ or $|u_2| \leq (k+1 - \text{m}(\bar{t}/u_1))d$.

We then define the *top* of a term \bar{t} , which is, intuitively, the only part of \bar{t} which can be used in a marked derivation using marks not greater than k .

Definition 13 (Top of a term). $\forall \bar{t} \in \mathcal{T}((\mathcal{F} \cup Q)^{\leq k}, \{\square\})$: $\text{Top}(\bar{t}) = \text{Red}(\bar{t}, \text{Topd}(\bar{t}))$.

Lemma 7 (projecting one step of \mathcal{R} on $S \cup \mathcal{A}$)

Let $\bar{s}, \bar{t} \in \mathcal{T}(\mathcal{F}^{\leq k})$ such that:

1- $\bar{s} \circ \rightarrow_{\mathcal{R}} \bar{t}$,

2- The marked rule (\bar{l}, r) used in the above rewriting-step is such that $\text{m}(\bar{l}) = 0$.

3- \bar{s} is m -increasing.

Then, $\text{Top}(\bar{s}) \rightarrow_{\mathcal{A}}^* \rightarrow_S \text{Top}(\bar{t})$.

Proof (Sketch) By the hypotheses of the lemma

$$\bar{s} = \overline{C[\bar{l}\bar{\sigma}]}, \quad \bar{t} = \overline{C[r\bar{\sigma}]}$$

for some $\overline{C}, \bar{\sigma}, \bar{l}, r, \bar{\sigma}$ fulfilling (24) of marked rewriting and $\text{m}(\bar{l}) = 0$. Let us then define a context \overline{D} and marked substitutions $\bar{\tau}, \bar{\tau}'$ by: for every $x \in \mathcal{V}$

$$\overline{D}[\] = \text{Top}(\overline{C}[\]). \quad (10)$$

$$x\bar{\tau} = \text{Top}(x\bar{\sigma}), \quad x\bar{\tau}' = \text{Red}(x\bar{\sigma}, \text{Pos}(x\bar{\tau})). \quad (11)$$

We claim that

$$\text{Top}(\bar{s}) \rightarrow_{\mathcal{A}}^* \overline{D}[\bar{l}\bar{\tau}] \rightarrow_S \overline{D}[\bar{l}\bar{\tau}'] = \text{Top}(\bar{t}). \quad (12)$$

(This claim is carefully checked in [10] and makes use of lemma 2).

Lemma 8 (projecting \mathcal{R} on $S \cup \mathcal{A}$)

Let $\bar{s}, \bar{t} \in \mathcal{T}(\mathcal{F}^{\leq k})$ and assume that \bar{s} is m -increasing. If $\bar{s} \stackrel{k}{\circ} \rightarrow_{\mathcal{R}} \bar{t}$ then, there exist terms $\bar{s}', \bar{t}' \in \mathcal{T}((\mathcal{F} \cup Q)^{\leq k})$ such that $\bar{s} \rightarrow_{\mathcal{A}}^* \bar{s}'$, $\bar{s}' \rightarrow_{S \cup \mathcal{A}}^* \bar{t}'$ and $\bar{t} \rightarrow_{\mathcal{A}}^* \bar{t}'$.

Proof. The marked derivation $\bar{s} \circ \rightarrow_{\mathcal{R}}^* \bar{t}$ is wbu, hence it can be decomposed into n successive steps where the hypothesis 2 of Lemma 7 is valid. Hypothesis 3 of Lemma 7 will also hold, owing to our assumption and to Lemma 5. We can thus deduce, inductively, from the conclusion of Lemma 7 that $\text{Top}(\bar{s}) \rightarrow_{S \cup \mathcal{A}}^* \text{Top}(\bar{t})$. The choice $\bar{s}' := \text{Top}(\bar{s}), \bar{t}' := \text{Top}(\bar{t})$ fulfills the conclusion of the lemma.

Lemma 9. *Let $s \in \mathcal{T}(\mathcal{F})$. Then $s \xrightarrow{k \rightarrow_{\mathcal{R}}^*} T$ iff $s \rightarrow_{S \cup \mathcal{A}}^* Q_f^{\leq k}$.*

Proof. (\Rightarrow): Suppose $s \xrightarrow{k \rightarrow_{\mathcal{R}}^*} t$ and $t \in T$. Let us consider the corresponding marked derivation

$$\bar{s} \xrightarrow{k \circ \rightarrow_{\mathcal{R}}^*} \bar{t} \quad (13)$$

where $\bar{s} := s$. Derivation (13) is wbu and lies in $\mathcal{T}(\mathcal{F}^{\leq k})$. Let us consider the terms \bar{s}', \bar{t}' given by Lemma 8:

$$\bar{s} \rightarrow_{\mathcal{A}}^* \bar{s}' \rightarrow_{S \cup \mathcal{A}}^* \bar{t}' \quad (14)$$

and $\bar{t} \rightarrow_{\mathcal{A}}^* \bar{t}'$. Since $\bar{t} \rightarrow_{\mathcal{A}}^* Q_f^{\leq k}$, by Lemma 2

$$\bar{t}' \rightarrow_{\mathcal{A}}^* Q_f^{\leq k}. \quad (15)$$

Combining (14) and (15) we obtain that $s \rightarrow_{S \cup \mathcal{A}}^* Q_f^{\leq k}$.

(\Leftarrow): Suppose $s \rightarrow_{S \cup \mathcal{A}}^* q^j \in Q_f^{\leq k}$. The hypotheses of Lemma 6 are met by $\bar{s} := s, \bar{s}' := s$ and $\bar{t} := q^j$. By Lemma 6 there exists some $\bar{t}' \in \mathcal{T}((\mathcal{F} \cup Q)^{\leq k})$ such that $s \xrightarrow{k \circ \rightarrow_{\mathcal{R}}^*} \bar{t}' \rightarrow_{\mathcal{A}}^* q^j \in Q_f^{\leq k}$. These derivations are mapped (by removal of the marks) into: $s \xrightarrow{k \rightarrow_{\mathcal{R}}^*} t' \rightarrow_{\mathcal{A}}^* q \in Q_f$, which shows that $t' \in T$ hence that $s \xrightarrow{k \rightarrow_{\mathcal{R}}^*} T$.

Proof. (of Theorem 2). By Lemma 9 ($k \rightarrow_{\mathcal{R}}^*$)[T] = $(\rightarrow_{S \cup \mathcal{A}}^*)[Q_f^{\leq k}] \cap \mathcal{T}(\mathcal{F})$. The rewriting systems \mathcal{S} and \mathcal{A} being ground are inverse-recognizability preserving (Theorem 1). So $(\rightarrow_{S \cup \mathcal{A}}^*)[Q_f^{\leq k}]$ is recognizable and thus $(k \rightarrow_{\mathcal{R}}^*)[T]$ is recognizable. In a second step one can extend Section 4.1 to the case where the restrictions (7) are not assumed anymore and consequently fully prove Theorem 2.

Corollary 1. *Every linear rewriting system of the class BU is inverse-recognizability preserving.*

4.2 Complexity

Upper-bounds. We estimate here the complexity of the algorithm underlying our proof of Theorem 2.

Theorem 3. *Let \mathcal{F} be a signature with symbols of arity ≤ 1 , let \mathcal{A} be some n.f.t.a recognizing a language $T \subseteq \mathcal{T}(\mathcal{F})$ and let \mathcal{R} be a finite rewriting system in $\text{BU}^-(1)$. One can compute a n.f.t.a \mathcal{B} recognizing $(\rightarrow_{\mathcal{R}}^*)[T]$ in time $O((\log(|\mathcal{F}|) \cdot \|\mathcal{A}\| \cdot \|\mathcal{R}\|)^3)$.*

Our proof consists in reducing the above problem, via the computation of the ground system \mathcal{S} of Section 4.1 to the computation of a set of descendants modulo some set of cancellation rules, which is achieved in cubic time in [2]. Since every left-basic semi-Thue system can be viewed as a $\text{BU}^-(1)$ term rewriting system, Theorem 3 extends [2]

(where a cubic complexity is proved for *cancellation systems* over a *fixed* alphabet) and improves [11] (where a degree 4 complexity is proved for *basic semi-Thue systems*).

Let us turn now to term rewriting systems over arbitrary signatures. Given a system \mathcal{R} we define

$$A(\mathcal{R}) := \max\{\text{Card}(\mathcal{P}\text{os}_V(l)) \mid l \rightarrow r \in \mathcal{R}\}.$$

Theorem 4. *Let $(\mathcal{R}, \mathcal{F})$ a finite rewriting system in $\text{BU}^-(1)$ and \mathcal{A} be some n.f.t.a over \mathcal{F} recognizing a language $T \subseteq \mathcal{T}(\mathcal{F})$. One can compute a n.f.t.a \mathcal{B} recognizing $(\rightarrow_{\mathcal{R}}^*)[T]$ in time polynomial w.r.t. $\|\mathcal{R}\| \cdot \|\mathcal{A}\|^{A(\mathcal{R})}$.*

Our proof consists in computing the ground system \mathcal{S} of Section 4.1 and to apply the result of [8], showing that the set of descendants of a recognizable set via a ground system \mathcal{S} can be achieved in polynomial time.

Lower-bound

Theorem 5. *There exists a fixed signature \mathcal{F} and two fixed recognizable sets T_1, T_2 over \mathcal{F} such that: the problem to decide, for a given linear term rewriting system $(\mathcal{R}, \mathcal{F})$ in $\text{BU}^-(1)$, whether $T_2 \cap (\rightarrow_{\mathcal{R}}^*)[T_1] \neq \emptyset$, is NP-hard.*

The proof in [10] consists of a P-time reduction of the problem SAT to the above problem. This result shows that the exponential upper-bound in Theorem 4 cannot presumably be significantly improved.

5 Strongly Bottom-Up Systems

The following theorem is established in [10], even in the restricted case of semi-Thue systems.

Theorem 6. *For every $k \geq 1$, the problem to determine whether a finite linear term rewriting system $(\mathcal{R}, \mathcal{F})$ is $\text{BU}(k)$ (resp. $\text{BU}^-(k)$), is undecidable.*

We are thus lead to define some stronger but *decidable* conditions.

5.1 Strongly Bottom-Up Systems

We abbreviate strongly bottom-up to sbu.

Definition 14. *A system $(\mathcal{R}, \mathcal{F})$ is said $\text{SBU}(k)$ iff for every $s \in \mathcal{T}(\mathcal{F}), \bar{t} \in \mathcal{T}(\mathcal{F}^{\leq k}), s \circ \rightarrow_{\mathcal{R}}^* \bar{t} \Rightarrow \bar{t} \in \mathcal{T}(\mathcal{F}^{\leq k})$.*

We denote by $\text{SBU}(k)$ the class of $\text{SBU}(k)$ systems and by $\text{SBU} = \bigcup_{k \in \mathbb{N}} \text{SBU}(k)$ the class of strongly bottom-up systems.

The following lemma is obvious.

Lemma 10. *Every $\text{SBU}(k)$ system is $\text{BU}(k)$.*

This stronger condition over term rewriting systems is interesting because of the following property.

Proposition 1. *For every $k \geq 0$, it is decidable whether a finite term rewriting system $(\mathcal{R}, \mathcal{F})$ is $\text{SBU}(k)$.*

Proof. Note that every marked derivation starting from some $s \in \mathcal{T}(\mathcal{F})$ and leading to some $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}) \setminus \mathcal{T}(\mathcal{F}^{\leq k})$ must decompose as $s \xrightarrow{k+1 \circ}^*_{\mathcal{R}} \bar{s}' \circ \xrightarrow{*}_{\mathcal{R}} \bar{t}$, with $\bar{s}' \in \mathcal{T}(\mathcal{F}^{\leq k+1}) \setminus \mathcal{T}(\mathcal{F}^{\leq k})$. A necessary and sufficient condition for \mathcal{R} to be $\text{SBU}(k)$ is thus that:

$$((\xrightarrow{k+1 \circ}^*_{\mathcal{R}})[\mathcal{T}(\mathcal{F}^{\leq k+1}) \setminus \mathcal{T}(\mathcal{F}^{\leq k})]) \cap \mathcal{T}(\mathcal{F}) = \emptyset. \quad (16)$$

By Theorem 2 the left-handside of equality (16) is a recognizable set for which we can construct a *f.t.a*; we then just have to test whether this *f.t.a* recognizes the empty set.

5.2 Sufficient Condition

We show here that the LFPO^{-1} condition of [25] is a *sufficient* and tractable condition for the SBU property. Let us associate with every rewriting system a *graph* whose vertices are the rules of the system and whose arcs (R, R') express some kind of overlap between the right-handside of R and the left-handside of R' . Every arc has a *label* in $\{a, b, c, d\}$ indicating the category of overlap that occurs and a *weight* which is an integer (0 or 1). The intuitive meaning of the weight is that any derivation step using the corresponding overlap might increase some mark by this weight. (This graph is a slight modification of the sticking-out graph of [25]).

Definition 15. *Let $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \setminus \mathcal{V}$ and $w \in \mathcal{Pos}_{\mathcal{V}}(t)$. We say that s sticks out of t at w if*

- 1- $\forall v \in \mathcal{Pos}(t)$ s.t. $\varepsilon \preceq v \prec w$, $v \in \mathcal{Pos}(s)$ and $s(v) = t(v)$.
- 2- $w \in \mathcal{Pos}(s)$ and $s/w \notin \mathcal{T}(\mathcal{F})$.

If in addition $s/w \notin \mathcal{V}$ then s strictly sticks out of t at w .

Definition 16. *Let $\mathcal{R} = \{l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n\}$ be a system. The sticking-out graph is the directed graph $\text{SG}(\mathcal{R}) = (V, E)$ where $V = \{1, \dots, n\}$ and E is defined as follows:*

- a) if l_j strictly sticks out of a subterm of r_i at w , $i \xrightarrow{a} j \in E$;
- b) if a strict subterm of l_j strictly sticks out of r_i at w , $i \xrightarrow{b} j \in E$;
- c) if a subterm of r_i sticks out of l_j at w , $i \xrightarrow{c} j \in E$;
- d) if r_i sticks out of a strict subterm of l_j at w , $i \xrightarrow{d} j \in E$.

Figure 4 shows all the possibilities in the four categories a, b, c, d . The *weight* of an arc of $\text{SG}(\mathcal{R})$ is defined by its label: if the label of the arc is a or b (resp. c or d) then its weight is 1 (resp. 0). The *weight of a path* in the graph is the sum of the weights of its arcs. The *weight of a graph* is the l.u.b. of the set of weights of all paths in the graph. The sticking-out graph of \mathcal{R}_1 of Example 1 is displayed in Figure 5. For an example with arities strictly greater than 1 see Example 6.

Proposition 2. *Let \mathcal{R} be a linear system. If $W(\text{SG}(\mathcal{R})) = k$ then $\mathcal{R} \in \text{SBU}(k+1)$.*

The proof consists of two steps. One first shows that the proposition holds for semi-Thue systems. In a second step one associates to every term-rewriting system \mathcal{R} its

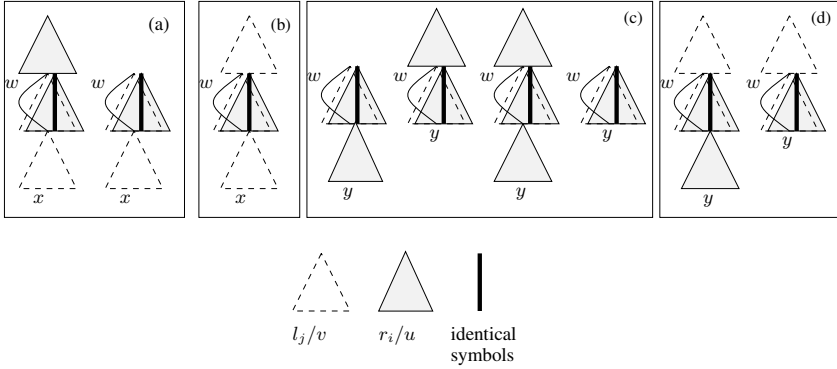


Fig. 4. Sticking-out cases



Fig. 5. The sticking-out graph of \mathcal{R}_1

branch semi-Thue system defined by: $T := \{u \rightarrow v \in \mathcal{F}^* \times \mathcal{F}^* \mid \exists l \rightarrow r \in \mathcal{R}, \exists x \in \mathcal{V}, ux \in \mathcal{B}(l), vx \in \mathcal{B}(r)\}$, where $\mathcal{B}(t)$ denotes the set of words over $\mathcal{F} \cup \mathcal{V}$ labeling the branches of t . The apparition of the mark $k + 1$ in a marked derivation for \mathcal{R} would imply the apparition of the mark $k + 1$ in a marked derivation for T . But every path of weight $k + 1$ in $SG(T)$ leads to a path of weight k in $SG(\mathcal{R})$.

Corollary 2 (sufficient condition). *Let \mathcal{R} be a linear system. If $W(SG(\mathcal{R}))$ is finite then $\mathcal{R} \in \text{SBU}$.*

The above sufficient condition can be tested in P-time. An immediate consequence of Corollary 2 is the following.

- Proposition 3.**
- 1- Every right-ground system is $\text{SBU}(0)$.
 - 2- Every inverse of a left-basic semi-Thue system is $\text{SBU}(1)$.
 - 3- Every growing linear system is $\text{SBU}(1)$.
 - 4- Every LFPO^{-1} system is SBU .

It can be checked, with ad hoc examples, that inclusions (2,3,4) above are strict.

Example 6. Let $\mathcal{R}_5 = \{f(g(x), a) \rightarrow f(x, b)\}$. $\mathcal{R}_5 \notin \text{LFPO}^{-1}$ as $SG(\mathcal{R})$ contains a loop a so a loop of weight $[1]$. It is easy to show by an ad-hoc proof that $\mathcal{R}_5 \in \text{SBU}^-(1)$. However our sufficient condition is not able to capture \mathcal{R}_5 .

6 Related Work and Perspectives

Related work Beside the references already mentioned, our work is also related to:

- [16, 15] from which we borrow some framework concerning equivalence relations over

derivations in order to give in [10] a criterium for the $BU(k)$ property for STS, - [15], where an undecidability result concerning a notion analogous to our SBU systems, makes us think that the SBU-condition (without any specified value of k) should be undecidable for semi-Thue systems, hence for term rewriting systems.

Perspectives Let us mention some natural perspectives of development for this work: - it is tempting to extend the notion of *bottom-up* rewriting (resp. system) to non-linear systems. This class would extend the class of growing systems studied in [19]. Also allowing free variables in the right-handsides seems reasonable. - a dual notion of *top-down* rewriting and a corresponding class of top-down systems should be defined. This class would presumably extend the class of Layered Transducing systems [22].

Some work in these two directions has been undertaken by the authors.

Acknowledgments. We thank the referees for their useful comments.

References

1. Benois, M.: Descendants of regular language in a class of rewriting systems: algorithm and complexity of an automata construction. In: Lescanne, P. (ed.) *Rewriting Techniques and Applications*. LNCS, vol. 256, pp. 121–132. Springer, Heidelberg (1987)
2. Benois, M., Sakarovitch, J.: On the complexity of some extended word problems defined by cancellation rules. *Inform. Process. Lett.* 23(6), 281–287 (1986)
3. Book, R.V., Jantzen, M., Wrathall, C.: Monadic Thue systems. *TCS* 19, pp. 231–251 (1982)
4. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: *Tree automata techniques and applications* (2002) Draft available from www.grappa.univ-lille3.fr/tata
5. Cremanns, R., Otto, F.: Finite derivation type implies the homological finiteness condition FP_3 . *J. Symbolic Comput.* 18(2), 91–112 (1994)
6. Dauchet, M., Heuillard, T., Lescanne, P., Tison, S.: Decidability of the confluence of finite ground term rewrite systems and of other related term rewrite systems. *Inf. Comput.* 88(2), 187–201 (1990)
7. Dauchet, M., Tison, S.: The theory of ground rewrite systems is decidable. In: *Fifth Annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, pp. 242–248. IEEE Comput. Soc. Press, Los Alamitos, CA (1990)
8. Deruyver, A., Gilleron, R.: The reachability problem for ground TRS and some extensions. In: Díaz, J., Orejas, F. (eds.) *TAPSOFT 1989*. LNCS, vol. 351, pp. 227–243. Springer, Heidelberg (1989)
9. Durand, I., Middeldorp, A.: Decidable call-by-need computations in term rewriting. *Information and Computation* 196, 95–126 (2005)
10. Durand, I., Sénizergues, G.: *Bottom-up rewriting for words and terms* (2007). Manuscript available at <http://dept-info.labri.u-bordeaux.fr/~ges>
11. Fülöp, Z., Jurvanen, E., Steinby, M., Vágvolgyi, S.: On one-pass term rewriting. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) *MFCS 1998*. LNCS, vol. 1450, pp. 248–256. Springer, Heidelberg (1998)
12. Geser, A., Hofbauer, D., Waldmann, J.: Match-bounded string rewriting systems. *Journal of Applied Algebra in Engineering, Communication and Computing* 15(3-4), 149–171 (2004)
13. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, Springer, Heidelberg (2005)

14. Jacquemard, F.: Decidable approximations of term rewriting systems. In: Ganzinger, H. (ed.) Proceedings of the 7th International Conference on Rewriting Techniques and Applications. LNCS, vol. 1103, pp. 362–376. Springer, Heidelberg (1996)
15. Knapik, T., Calbrix, H.: Thue specifications and their monadic second-order properties. *Fund. Inform.* 39(3), 305–325 (1999)
16. Lafont, Y., Prouté, A.: Church-Rosser property and homology of monoids. *Math. Structures Comput. Sci.* 1(3), 297–326 (1991)
17. Lohrey, M., Sénizergues, G.: Rational subsets of HNN-extensions (2005) Manuscript available at <http://dept-info.labri.u-bordeaux.fr/~ges>
18. Silva, P.V., Kambites, M., Steinberg, B.: On the rational subset problem for groups. *J. of Algebra* (To appear)
19. Nagaya, T., Toyama, Y.: Decidability for left-linear growing term rewriting systems. *Information and Computation* 178(2), 499–514 (2002)
20. Réty, P., Vuotto, J.: Tree automata for rewrite strategies. *J. Symb. Comput.* 40(1), 749–794 (2005)
21. Sakarovitch, J.: *Syntaxe des langages de Chomsky, essai sur le déterminisme*. Thèse de doctorat d'état de l'université Paris VII, pp. 1–175 (1979)
22. Seki, H., Takai, T., Fujinaka, Y., Kaji, Y.: Layered transducing term rewriting system and its recognizability preserving property. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, Springer, Heidelberg (2002)
23. Sénizergues, G.: Formal languages & word-rewriting. In: Comon, H., Jouannaud, J.-P. (eds.) *Term rewriting (Font Romeu, 1993)*. LNCS, vol. 909, pp. 75–94. Springer, Heidelberg (1995)
24. Seynhaeve, F., Tison, S., Tommasi, M.: Homomorphisms and concurrent term rewriting. In: FCT, pp. 475–487 (1999)
25. Takai, T., Kaji, Y., Seki, H.: Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. *Scientiae Mathematicae Japonicae* (To appear, preliminary version: IEICE Technical Report COMP98-45) (2006)

Adjunction for Garbage Collection with Application to Graph Rewriting^{*}

D. Duval¹, R. Echahed², and F. Prost²

¹ LJK

B. P. 53, F-38041 Grenoble, France

Dominique.Duval@imag.fr

² LIG

46, av Félix Viallet, F-38031 Grenoble, France

Rachid.Echahed@imag.fr, Frederic.Prost@imag.fr

Abstract. We investigate garbage collection of unreachable parts of rooted graphs from a *categorical* point of view. First, we define this task as the right adjoint of an inclusion functor. We also show that garbage collection may be stated via a left adjoint, hence preserving colimits, followed by two right adjoints. These three adjoints cope well with the different phases of a traditional garbage collector. Consequently, our results should naturally help to better formulate graph transformation steps in order to get rid of garbage (unwanted nodes). We illustrate this point on a particular class of graph rewriting systems based on a double pushout approach and featuring edge redirection. Our approach gives a neat rewriting step akin to the one on terms, where garbage never appears in the reduced term.

1 Introduction

Garbage collection has been introduced [5,9] in order to improve the management of memory space dedicated to the run of a process. Such memory can be seen as a rooted graph, where the nodes reachable from the roots represent the memory cells whose content can potentially contribute to the execution of the process, while the unreachable nodes represent the garbage, i.e., the cells that may become allocated at will when memory is required. Several algorithms have been proposed in the literature in order to compute the reachable and unreachable nodes (see for instance [4,8]).

In this paper, we investigate garbage collection from a *categorical* point of view. More precisely, our main purpose is a theoretical definition of the process of calculating the reachable nodes of a rooted graph. This corresponds to removing the garbage, in the sense of calculating the memory space made of the reachable cells, starting from a memory space that may include reachable as well as unreachable cells. We show that this process can be defined as the right

^{*} This work has been partly funded by the projet ARROWS of the French *Agence Nationale de la Recherche*.

adjoint of an inclusion functor. We also propose an alternative definition of this process via a left adjoint, followed by two right adjoints. This second definition is close to the actual *tracing* garbage collection algorithms, which proceed by marking the reachable nodes before sweeping the garbage.

Besides the categorical characterisation of the garbage collection process, which can be considered as a motivation per se, the original motivation of this work comes from the definition of graph rewrite steps based on the double pushout approach [7]. In these frameworks a rewrite step is defined as a span $L \leftarrow K \rightarrow R$ where L , K and R are graphs and the arrows represent graph homomorphisms. Let us consider, for instance, the category of graphs defined in [6] (this category is similar to the category **Gr** of section 3.1 except that graphs are not rooted).

The application of such a rule to a graph G consists in finding a homomorphism (a matching) $m : L \rightarrow G$ and computing the reduced graph H so that the following diagram

$$\begin{array}{ccccc}
 L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 m \downarrow & & \downarrow d & & \downarrow m' \\
 G & \xleftarrow{l'} & D & \xrightarrow{r'} & H
 \end{array}$$

constitutes a double pushout (some conditions are required to ensure the existence of this double pushout [7]).

A main drawback of this approach is that the reduced graph H may contain unreachable nodes. To illustrate this point, let us consider a simple example. Let $f(x) \rightarrow f(b)$ be a classical term rewrite rule. When it is applied on the term $f(a)$, the resulting term is $f(b)$. However, in the double pushout approach (where terms are viewed as graphs), the reduced graph is not just $f(b)$: it includes also the constant a . Indeed, the term rewrite rule $f(x) \rightarrow f(b)$ is turned into a span $f(x) \leftarrow K_0 \rightarrow f(b)$, where the arrows are graph homomorphisms (the content of K_0 does not matter here). When this rule is applied to the graph $f(a)$, according to the double pushout approach, we get the diagram

$$\begin{array}{ccccc}
 f(x) & \xleftarrow{\quad} & K_0 & \xrightarrow{\quad} & f(b) \\
 \downarrow & & \downarrow & & \downarrow \\
 f(a) & \xleftarrow{\quad} & D_0 & \xrightarrow{\quad} & H_0
 \end{array}$$

where H_0 contains both terms $f(b)$ and a . Actually, the element a occurs in D_0 , because the left-hand side is a pushout; thus, a also occurs in H_0 , because the right-hand side is a pushout. In order to get rid of a , and more generally to remove all unreachable nodes from the graph H , we propose to use our categorical approach of garbage collection.

When graph rewrite steps are defined following an algorithmic approach such as [2], garbage is easily incorporated within a rewrite step. Unfortunately, in categorical approaches to graph rewriting, garbage is not easily handled. In section 8 of [1], Banach discussed the problem of garbage in an abstract way. He mainly considered what is called “garbage retention”, that is to say, garbage is not removed from a graph, as we do, but it should not participate in the rewriting process. In [3], Van den Broek discussed the problem of generated garbage

in the setting of graph transformation based on single pushout approach. He introduced the notion of proper graphs. Informally, a proper graph is a graph where garbage cannot be reachable from non garbage part. The rewrite relation is defined as a binary relation over proper graphs. That is to say, a rewrite rule can be fired only if the resulting graph is proper. In some sense, Van den Broek performs a kind of garbage retention as Banach does.

Outline of the paper

Rooted graphs are introduced in section 2 in order to model reachable and unreachable parts in data-structures. Garbage collection is presented in terms of right and left adjoints in section 3. A double pushout approach for rooted graphs rewriting is defined in section 4, and garbage collection is incorporated within this rewriting setting. We conclude in section 5. Due to space limitations, proofs have been omitted.

2 Rooted Graphs

Rooted graphs are defined as term graphs [2], together with a subset of nodes called roots. These graphs are intended to model usual data-structures implemented with pointers, such as circular lists. The addition of roots models the fact that, due to pointer redirections, some data may become unreachable.

Definition 1 (Signature). A *signature* Ω is a set of operation symbols such that each operation symbol f in Ω is provided with a natural number $\text{ar}(f)$ called its *arity*.

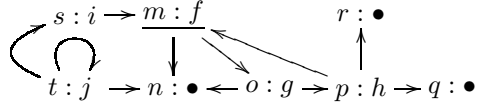
We assume Ω fixed throughout the rest of the paper. Moreover, for each set A , the set of strings over A is denoted A^* , and for each map $f : A \rightarrow B$, the map $f^* : A^* \rightarrow B^*$ is the extension of f to strings defined by $f^*(a_1 \dots a_n) = f(a_1) \dots f(a_n)$.

Definition 2 (Graph). A *rooted graph* is a tuple $G = (\mathcal{N}_G, \mathcal{N}_G^R, \mathcal{N}_G^\Omega, \mathcal{L}_G, \mathcal{S}_G)$ where \mathcal{N}_G is the set of *nodes* of G , $\mathcal{N}_G^R \subseteq \mathcal{N}_G$ is the set of *roots*, $\mathcal{N}_G^\Omega \subseteq \mathcal{N}_G$ is the set of *labeled nodes*, $\mathcal{L}_G : \mathcal{N}_G^\Omega \rightarrow \Omega$ is the *labeling function*, and $\mathcal{S}_G : \mathcal{N}_G^\Omega \rightarrow \mathcal{N}_G^*$ is the *successor function* such that, for each labeled node n , the length of the string $\mathcal{S}_G(n)$ is the arity of the operation $\mathcal{L}_G(n)$.

The *arity* of a node n is the arity of its label and the i -th successor of a node n is denoted $\text{succ}_G(n, i)$. The *edges* of a graph G are the pairs (n, i) where $n \in \mathcal{N}_G^\Omega$ and $i \in \{1, \dots, \text{ar}(n)\}$, the *target* is the node $\text{tgt}(n, i) = \text{succ}_G(n, i)$. The set of edges of G is written \mathcal{E}_G . The fact that $f = \mathcal{L}_G(n)$ is written $n : f$, an unlabeled node n of G is written $n : \bullet$. Informally, one may think of \bullet as an anonymous variable. The set of unlabeled nodes of G is denoted $\mathcal{N}_G^\mathcal{X}$, so that $\mathcal{N}_G = \mathcal{N}_G^\Omega + \mathcal{N}_G^\mathcal{X}$, where “+” stands for the disjoint union.

Example 1. Let G be the graph defined by $\mathcal{N}_G = \{m, n, o, p, q, r, s, t\}$, $\mathcal{N}_G^\Omega = \{m, o, p, s, t\}$, $\mathcal{N}_G^X = \{n, q, r\}$, \mathcal{L}_G is defined by: $[m \mapsto f, o \mapsto g, p \mapsto h, s \mapsto i, t \mapsto j]$, \mathcal{S}_G is defined by: $[m \mapsto no, o \mapsto np, p \mapsto qrm, s \mapsto m, t \mapsto tsn]$, and roots of G are $\mathcal{N}_G^R = \{m\}$.

Graphically G is represented as:

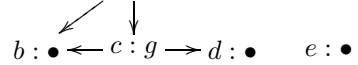


Roots of graphs are underlined. Generally in our examples the order of successors is either irrelevant or clear from the context.

Definition 3 (Graph homomorphism). A *rooted graph homomorphism* $\varphi : G \rightarrow H$ is a map $\varphi : \mathcal{N}_G \rightarrow \mathcal{N}_H$ that preserves the roots, the labeled nodes and the labeling and successor functions, i.e., $\varphi(\mathcal{N}_G^R) \subseteq \mathcal{N}_H^R$, $\varphi(\mathcal{N}_G^\Omega) \subseteq \mathcal{N}_H^\Omega$, and for each labeled node n , $\mathcal{L}_H(\varphi(n)) = \mathcal{L}_G(n)$ and $\mathcal{S}_H(\varphi(n)) = \varphi^*(\mathcal{S}_G(n))$.

The image $\varphi(n, i)$ of an edge (n, i) of G is defined as the edge $(\varphi(n), i)$ of H .

Example 2. Consider the following graph H : $v : i \rightarrow \underline{a : f}$



Let $\varphi : \mathcal{N}_H \rightarrow \mathcal{N}_G$, where G is the graph in Example 1, be defined as: $\varphi = [a \mapsto m, b \mapsto n, c \mapsto o, d \mapsto p, e \mapsto p, v \mapsto s]$. Then, φ is a graph homomorphism from H to G .

3 Garbage Collection and Adjunction

A node in a rooted graph is *reachable* if it is a descendant of a root; the unreachable nodes form the *garbage* of the graph. We now address the problem of garbage collection in graphs, in both its aspects: either removing the unreachable nodes or reclaiming them; we also consider the marking of reachable nodes, which constitutes a major step in the so-called *tracing* garbage collection process. We prove in this section that the tracing garbage collection process can be easily expressed in term of adjunctions.

3.1 Garbage Removal Is a Right Adjoint

Rooted graphs and their homomorphisms form the *category of rooted graphs*. From now on, in this paper, the category of rooted graphs is denoted \mathbf{Gr} , the category of non-rooted graphs is denoted \mathbf{Gr}_0 , and by “graph” we mean “rooted graph”, unless explicitly stated.

Definition 4 (Reachable nodes). The *reachable nodes* of a graph are defined recursively, as follows: a root is reachable, and the successors of a reachable node are reachable nodes.

Example 3. The graph G defined in example 1, has m, n, o, p, q, r as reachable nodes. Nodes s, t are considered as garbage, as well as the edges out of these nodes.

Definition 5 (Reachable graph). A *reachable graph* is a graph where all nodes are reachable.

The reachable graphs and the graph homomorphisms between them form a full subcategory of \mathbf{Gr} , called the *category of reachable graphs*, \mathbf{RGr} . Let V denote the inclusion functor: $V : \mathbf{RGr} \rightarrow \mathbf{Gr}$.

Definition 6 (Maximal reachable subgraph). The *maximal reachable subgraph* of G is the graph $\Lambda(G)$ such that $\mathcal{N}_{\Lambda(G)}$ is the set of reachable nodes of G , $\mathcal{N}_{\Lambda(G)}^R = \mathcal{N}_G^R$, $\mathcal{N}_{\Lambda(G)}^\Omega = \mathcal{N}_{\Lambda(G)} \cap \mathcal{N}_G^\Omega$, and $\mathcal{L}_{\Lambda(G)}$, $\mathcal{S}_{\Lambda(G)}$ are the restrictions of \mathcal{L}_G , \mathcal{S}_G to $\mathcal{N}_{\Lambda(G)}$.

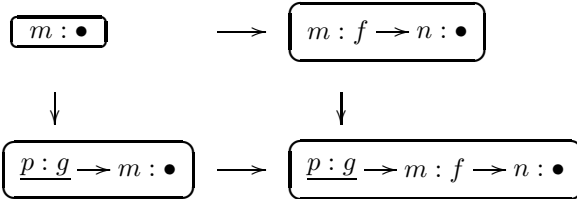
Since a graph homomorphism $\varphi : G \rightarrow H$ preserves the roots and the successors, it does also preserve the reachable nodes. Hence, it can be restricted to the maximal reachable subgraphs: $\Lambda(\varphi) : \Lambda(G) \rightarrow \Lambda(H)$.

Definition 7 (Garbage removal functor). The *garbage removal* is the functor: $\Lambda : \mathbf{Gr} \rightarrow \mathbf{RGr}$ that maps each graph G to its maximal reachable subgraph $\Lambda(G)$ and each graph homomorphism $\varphi : G \rightarrow H$ to its restriction $\Lambda(\varphi) : \Lambda(G) \rightarrow \Lambda(H)$.

Clearly, the composed functor $\Lambda \circ V$ is the identity of \mathbf{RGr} : a reachable graph is not modified under garbage removal. Moreover, the next result proves that \mathbf{RGr} is a *coreflective* full subcategory of \mathbf{Gr} .

Proposition 1 (Garbage removal is a right adjoint). The garbage removal functor Λ is the right adjoint for the inclusion functor V .

The garbage removal functor $\Lambda : \mathbf{Gr} \rightarrow \mathbf{RGr}$ is not a left adjoint. Indeed, a left adjoint does preserve the colimits, whereas the functor Λ does not preserve pushouts, as shown in the following example:



If Λ is applied to this square, the graphs of the upper row will be the empty graphs, whereas the graphs of the lower row will remain unchanged. The obtained diagram is no longer a pushout in category \mathbf{RGr} , since label f and node $n : \bullet$ appear from nowhere.

3.2 Reachability Marking Is a Left Adjoint

Definition 8 (Marked graph). A *marked graph* is a graph M with a set $\mathcal{N}_M^* \subseteq \mathcal{N}_M$ of *marked nodes*, such that every root is marked and every successor of a marked node is marked (so that all the reachable nodes of a marked graph are marked but unreachable nodes can be marked). A marked graph homomorphism is a graph homomorphism that preserves the marked nodes.

The marked graphs and their homomorphisms form the *category of marked graphs* \mathbf{Gr}' .

Let $\Delta : \mathbf{Gr}' \rightarrow \mathbf{Gr}$ denote the underlying functor, that forgets about the marking, and let $\nabla : \mathbf{Gr} \rightarrow \mathbf{Gr}'$ denote the functor that generates a marked graph from a graph, by marking all its reachable nodes. The next result is straightforward, since the marking does not modify the underlying graph.

Proposition 2 (Reachability marking is a left adjoint). The reachability marking functor ∇ is the left adjoint for the underlying functor Δ , and this adjunction is such that $\Delta \circ \nabla \cong \text{Id}_{\mathbf{Gr}}$.

Definition 9 (Reachable marked graph). A *reachable marked graph* is a marked graph where all nodes are reachable. So, all the nodes of a reachable marked graph are marked.

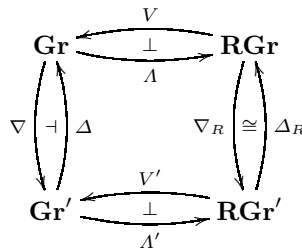
The reachable marked graphs and the marked graph homomorphisms form a full subcategory of \mathbf{Gr}' , called the *category of reachable marked graphs*, \mathbf{RGr}' . One can note that \mathbf{RGr}' is isomorphic to \mathbf{RGr} . We make the distinction between those two categories because it enables to give a neat categorical view of how garbage collection is performed. Let V' denote the inclusion functor: $V' : \mathbf{RGr}' \rightarrow \mathbf{Gr}'$. As in proposition 1, the inclusion functor V' has a right adjoint: $\Delta' : \mathbf{Gr}' \rightarrow \mathbf{RGr}'$ which is the garbage removal functor for marked graphs.

3.3 Tracing Garbage Collection

A *tracing* garbage collector first determines which nodes are reachable, and then either discards or reclaims all the unreachable nodes. In categorical terms, the fact that reachability marking can be used to perform garbage collection is expressed by theorem 1 and proposition 3 below.

Similarly to the adjunction $\nabla \dashv \Delta$, there is an adjunction $\nabla_R \dashv \Delta_R$ where $\Delta_R : \mathbf{RGr}' \rightarrow \mathbf{RGr}$ denotes the underlying functor, that forgets about the marking, and $\nabla_R : \mathbf{RGr} \rightarrow \mathbf{RGr}'$ denotes the functor that generates a marked graph from a reachable one, by marking all its nodes. Actually, this adjunction is an isomorphism. \mathbf{RGr} and \mathbf{RGr}' are not identified since they formalize well a phase of a natural garbage collector. Indeed, classical garbage collector begins by suspending the execution of current processes. Then it performs a marking of reachable memory cells (∇). Afterwards, all unmarked cells are deallocated (Δ') and finally the marking is forgotten (Δ_R).

In the following diagram, the four adjunction pairs are represented.



Functors $\nabla \circ V$ and $V' \circ \nabla_R$ are equal, since they both map a reachable graph H to the marked graph made of H with all its nodes marked:

$$\nabla \circ V = V' \circ \nabla_R : \mathbf{RGr} \rightarrow \mathbf{Gr}'$$

Theorem 1 below provides a categorical formalization of the tracing garbage removal process, which can be decomposed in three steps: First, the main step freely generates the marked graph $\nabla(G)$ from the given graph G . Then, the reachable marked graph $\Lambda'(\nabla(G))$ is obtained by throwing away the non-marked nodes. Finally, the reachable graph $\Delta_R(\Lambda'(\nabla(G)))$ is obtained by forgetting the marking.

According to theorem 1, $\Delta_R(\Lambda'(\nabla(G)))$ is isomorphic to $\Lambda(G)$.

Theorem 1 (Garbage removal through reachability marking)

$$\Lambda \cong \Delta_R \circ \Lambda' \circ \nabla .$$

So, the garbage removal functor Λ can be replaced by $\Delta_R \circ \Lambda' \circ \nabla$. This means that the main step in the tracing garbage removal process is the reachability marking: indeed, the application of Δ_R is “trivial”, and the application of Λ' to the image of ∇ simply throws away the unmarked nodes.

In order to express garbage reclaiming, let \mathcal{N} and \mathcal{N}^* denote the functors from \mathbf{Gr}' to \mathbf{Set} that map a marked graph to its set of nodes and to its set of marked nodes, respectively. They can be combined into one functor with values in the following category **SubSet**: the objects of *Subset* are the pairs of sets (X, Y) such that $Y \subseteq X$, and a morphism is a pair of maps (f, g) from (X, Y) to (X', Y') , where $f : X \rightarrow X'$, $g : Y \rightarrow Y'$, and g is the restriction of f . $(\mathcal{N}, \mathcal{N}^*) : \mathbf{Gr}' \rightarrow \mathbf{SubSet}$

The complement $X \setminus Y$ is the set of unreachable nodes: this is expressed in the next result. Note that the *set complement*, that maps (X, Y) to $X \setminus Y$, cannot reasonably be extended to a functor from **SubSet** to **Set**.

Proposition 3 (Garbage reclaiming through reachability marking)

The composition of ∇ with $(\mathcal{N}, \mathcal{N}^*)$, followed by the set complement, provides the set of unreachable nodes.

So, the garbage reclaiming can be expressed as $(\mathcal{N}, \mathcal{N}^*) \circ \nabla$ followed by the set complement. Here also, this means that the main step in the tracing garbage reclaiming process is the reachability marking.

4 Application: Rooted Graph Rewriting with Garbage Removal

In this section we focus on a class of graph rewrite systems dedicated to transform data-structures with pointers [6]. This class has been defined using the double pushout approach [7]. We mainly show how rewrite steps can be enhanced by integrating garbage removal. This integration can be generalized to other rewrite systems based on pushouts, thanks to the fact that left adjoints preserve colimits.

4.1 Disconnections

This section will be used in the left hand side of the double pushout construction. Definitions are adapted from [6], with a more homogeneous presentation.

The disconnection of a graph L is made of a graph K and a graph homomorphism $l : K \rightarrow L$. Roughly speaking, K is obtained by redirecting some edges of L toward new, unlabeled targets, and the homomorphism l reconnects all the disconnected nodes: \mathcal{N}_K is made of \mathcal{N}_L together with some new, unlabeled nodes, and l is the identity on \mathcal{N}_L .

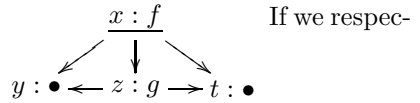
Definition 10 (Disconnection kit). A *disconnection kit* $k = (E_l, N_g, E_g)$ for a graph L , where E_l, E_g are subsets of \mathcal{E}_L and $N_g \subseteq \mathcal{N}_L$, is made of: (i) a set of edges E_l , called the *locally redirected edges*, (ii) a set of nodes N_g , called the *globally redirected nodes*, and (iii) another set of edges E_g , called the *globally redirected edges*, that is disjoint from E_l and such that the target of every edge in E_g is in N_g . A disconnection kit (E_l, N_g, \emptyset) is simply denoted (E_l, N_g) .

Definition 11 (Disconnection of a graph). Let L be a graph, with a disconnection kit $k = (E_l, N_g, E_g)$. Let K be the graph defined by:

- $\mathcal{N}_K = \mathcal{N}_L + \mathcal{N}_E + \mathcal{N}_N$, where \mathcal{N}_E is made of one new node $n[i]$ for each edge $(n, i) \in E_l$ and \mathcal{N}_N is made of one new node $n[0]$ for each node $n \in N_g$,
- \mathcal{N}_K^R is made of one node for each root n of L : n itself if $n \notin N_g$ and $n[0]$ if $n \in N_g$.
- $\mathcal{N}_K^\Omega = \mathcal{N}_L^\Omega$,
- for each $n \in \mathcal{N}_L^\Omega$: $\mathcal{L}_K(n) = \mathcal{L}_L(n)$,
- for each $n \in \mathcal{N}_L^\Omega$ and $i \in \{1, \dots, \text{ar}(n)\}$: if $(n, i) \notin E_l + E_g$ then $\text{succ}_K(n, i) = \text{succ}_L(n, i)$, if $(n, i) \in E_l$ then $\text{succ}_K(n, i) = n[i]$ and if $(n, i) \in E_g$ then $\text{succ}_K(n, i) = \text{tgt}(n, i)[0]$.

Let $l : \mathcal{N}_K \rightarrow \mathcal{N}_L$ be the map defined by: $l(n) = n$ if $n \in \mathcal{N}_L$, $l(n[i]) = \text{succ}_L(n, i)$ if $(n, i) \in E_l$, $l(n[0]) = n$ if $n \in N_g$. Clearly l preserves the roots, the labeled nodes and the labeling and successor functions, so that it is a graph homomorphism. Then $l : K \rightarrow L$ is the *disconnection of L with respect to k* .

Example 4. Let L_{\square} be the following graph:



tively consider disconnection kits $k_1 = (\{(x, 1), (z, 2)\}, \emptyset)$ and $k_2 = (\{(x, 3)\}, \{x\})$, we have the following disconnections of L_{\square} , respectively K_{\square} and K'_{\square} :



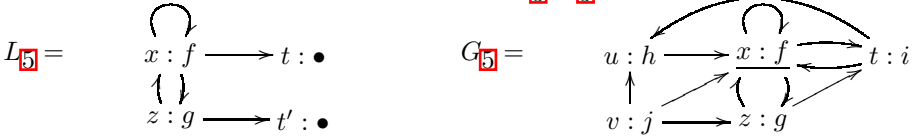
Definition 12 (Matching). Let L be a graph with a disconnection kit $k = (E_l, N_g)$. A *matching of L consistent with k* is a graph homomorphism $m : L \rightarrow G$ such that the restriction of m to $(\mathcal{N}_L^\Omega \cup N_g)$ is injective.

Definition 13 (Disconnection of a matching). Let L be a graph, with a disconnection kit $k = (E_l, N_g)$, and let $m : L \rightarrow G$ be a matching of L consistent with k . Let $E'_l = m(E_l)$ and $N'_g = m(N_g)$ (since m is a matching, the restrictions of m are bijections: $E_l \cong E'_l$ and $N_g \cong N'_g$). Let E'_g be the set of the edges of $G - m(L)$ with their target in N'_g , and let $k' = (E'_l, N'_g, E'_g)$. Let $l : K \rightarrow L$ be the disconnection of L with respect to k , and $l' : D \rightarrow G$ the disconnection of G with respect to k' . Let $d : \mathcal{N}_K \rightarrow \mathcal{N}_D$ be the map defined by: $d(n) = m(n)$ if $n \in \mathcal{N}_L$, $d(n[i]) = m(n)[i]$ if $n[i] \in \mathcal{N}_E$ and $d(n[0]) = m(n)[0]$ if $n[0] \in \mathcal{N}_N$. Clearly, d is a graph homomorphism. Then the following square in \mathbf{Gr} is called *the disconnection of m with respect to k* :

$$\begin{array}{ccc}
 L & \xleftarrow{l} & K & \xrightarrow{d} & D \\
 & & & \searrow^{l'} & \\
 & & & & G \\
 & & m & \swarrow &
 \end{array}$$

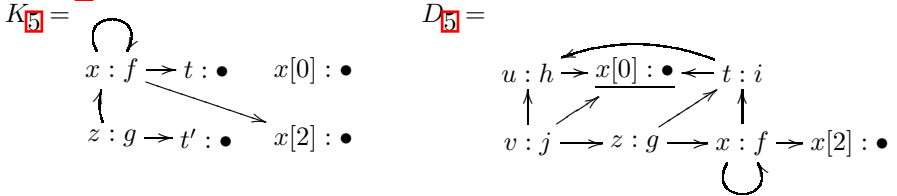
In other words the disconnection of a matching with respect to a disconnection kit consists in the building of, D (and the appropriate morphisms) once K, L, G and m, l are given. Informally D is made of three parts. The first one is the part of G that is not matched. The second part is the image of L in G . Finally there are nodes without labels introduced to perform redirections ($m(n)[i]$ for local redirections, $m(n)[0]$ for global ones).

Example 5. Consider the following graphs $L_{\mathfrak{F}}$, $G_{\mathfrak{F}}$:



and let k be the disconnection kit $(\{(x, 2)\}, \{x\})$, the edge $(x, 2)$ being the edge joining node x to node z .

The graph homomorphism $m_{\mathfrak{F}} = [x \mapsto x, t \mapsto t, t' \mapsto t, z \mapsto z]$, is a morphism from $L_{\mathfrak{F}}$ to $G_{\mathfrak{F}}$. It is also a matching of $L_{\mathfrak{F}}$ consistent with k . Now by disconnection of $G_{\mathfrak{F}}$ with respect to k we have the graphs $K_{\mathfrak{F}}$ which is the disconnection of $L_{\mathfrak{F}}$ with respect to disconnection kit $(\{(x, 2)\}, \{x\})$, and $D_{\mathfrak{F}}$ which is the disconnection of $G_{\mathfrak{F}}$ with respect to disconnection kit $(\{(x, 2)\}, \{x\}, \{(u, 1), (v, 2), (t, 2)\})$:



4.2 Rooted Graph Rewriting

Definition 14 (Rewrite rule). A *rewrite rule*, or *production*, is a span of graphs $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ where l is the disconnection of L with respect to a disconnection kit $k = (E_l, N_g)$, and the restriction of r to \mathcal{N}_L^X is injective and has its values in \mathcal{N}_R^X . Then p is a rewrite rule *consistent with k* .

A rewrite step is defined from a rewrite rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a matching $m : L \rightarrow G$, both with respect to a disconnection kit $k = (E_l, N_g)$ of L . The role of the rewrite step consists in: (i) adding to G an instance of the right-hand side R of p , (ii) performing some local redirections of edges in G : each edge (n, i) in $m(E_l)$ is redirected to the new target $n[i]$, (iii) performing some global redirections of edges in G : all incoming edges of a node n in $m(N_g)$, except the edges in the image of the matching, are redirected to the new target $n[0]$, (iv) modifying the roots of G : if n is a root in G not in $m(N_g)$ then it remains a root, but if n is a root in G and in $m(N_g)$ then $n[0]$ becomes a root instead of n .

As in [6], the basic ingredient in the double pushout approach to graph rewriting is lemma 1 below, about the reflection of pushouts by a faithful functor. The faithful functor in [6] was the node functor \mathcal{N} , from the category \mathbf{Gr}_0 of non-rooted graphs to the category of sets. Here, it will be the forgetful functor U_0 from the category \mathbf{Gr} of rooted graphs to the category \mathbf{Gr}_0 of non-rooted graphs, which clearly is faithful. Since this lemma has not been stated in this form in [6], its proof is given below.

Lemma 1 (Pushout reflection). Let $\Phi : \mathbf{A} \rightarrow \mathbf{A}'$ be a faithful functor. Let $\Sigma = (A_1 \xleftarrow{f_1} A_0 \xrightarrow{f_2} A_2)$ be a span and let Γ be a square in \mathbf{A} :

$$\begin{array}{ccccc} A_1 & & A_0 & & A_2 \\ & \swarrow f_1 & & \searrow f_2 & \\ & & A & & \\ & \searrow g_1 & & \swarrow g_2 & \end{array}$$

If $\Phi(\Gamma)$ is a pushout in \mathbf{A}' and if for each cocone Δ on Σ in \mathbf{A} , there is a morphism $h : A \rightarrow B$ in \mathbf{A} (where B is the vertex of Δ) such that $\Phi(h)$ is the cofactorisation of $\Phi(\Delta)$ with respect to $\Phi(\Gamma)$, then Γ is a pushout in \mathbf{A} .

For each span Σ of sets $(N_1 \xleftarrow{\varphi_1} N_0 \xrightarrow{\varphi_2} N_2)$, let \sim denote the equivalence relation induced by Σ on $N_1 + N_2$, which means that it is generated by $\varphi_1(n_0) \sim \varphi_2(n_0)$ for all $n_0 \in N_0$, and let N be the quotient $N = (N_1 + N_2) / \sim$. For $i \in \{1, 2\}$, let $\psi_i : N_i \rightarrow N$ map every node n_i of G_i to its class modulo \sim . Then, it is well-known that the following square is a pushout in \mathbf{Set} , which will be called *canonical*:

$$\begin{array}{ccccc} & & N_0 & & \\ & \swarrow \varphi_1 & & \searrow \varphi_2 & \\ N_1 & & & & N_2 \\ & \searrow \psi_1 & & \swarrow \psi_2 & \\ & & N & & \end{array}$$

The notion of “strongly labeled span of graphs” comes from [6], where it was defined for non-rooted graphs, but actually roots are not involved in this notion.

Definition 15 (Strongly labeled span of graphs). A span $\Sigma = (G_1 \xleftarrow{\varphi_1} G_0 \xrightarrow{\varphi_2} G_2)$ in \mathbf{Gr} is *strongly labeled* if, as soon as two labeled nodes in $\mathcal{N}(G_1) + \mathcal{N}(G_2)$ are equivalent with respect to Σ , they have the same label and equivalent successors.

Definition 16 (Canonical square of graphs). Let $\Sigma = (G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2)$ be a strongly labeled span in \mathbf{Gr} . The *canonical square on Σ* is the square in \mathbf{Gr} :

$$\begin{array}{ccccc}
 G_1 & \xleftarrow{\varphi_1} & G_0 & \xrightarrow{\varphi_2} & G_2 \\
 & \searrow \psi_1 & & \swarrow \psi_2 & \\
 & & G & &
 \end{array}$$

where the underlying square of nodes is the canonical pushout in **Set**, a node n in G is a root if and only if $n = \psi_i(n_i)$ for a root n_i in G_1 or G_2 , a node n in G is labeled if and only if $n = \psi_i(n_i)$ for a labeled node n_i in G_1 or G_2 , and moreover the label of n is the label of n_i and the successors of n are the equivalence classes of the successors of n_i .

Clearly, $\Gamma(\Sigma)$ is a commutative square in **Gr**. It is actually a pushout in **Gr** as stated in Theorem 2. The next result is proved in [6].

Lemma 2 (Pushout of non-rooted graphs). Let Σ be a strongly labeled span in **Gr**, and let Γ be the canonical square on Σ . Then $U_0(\Gamma)$ is a pushout in **Gr**₀.

Theorem 2 (Pushout of rooted graphs). Let Σ be a strongly labeled span in **Gr**, and let Γ be the canonical square on Σ . Then Γ is a pushout in **Gr**.

It is easy to see that a disconnection square is the canonical square on a strongly labeled span, so that the next result follows from theorem 2.

Theorem 3 (A pushout complement). Let L be a graph with a disconnection kit k and let m be a matching of L consistent with k . The disconnection square of m with respect to k is a pushout in the category of graphs.

Theorem 3 means that d and l' form a *complement pushout* to l and m . Other complement pushouts to l and m can be obtained by replacing E'_g , in definition 13, by any of its subsets.

The next result is not so easy, its proof can be found in [6] (except for the property of the roots, which is clear).

Theorem 4 (A direct pushout). Let p be a rewrite rule ($L \xleftarrow{l} K \xrightarrow{r} R$) and $m : L \rightarrow G$ a matching, both consistent with a disconnection kit k of L . Then the span $D \xleftarrow{d} K \xrightarrow{r} R$ is strongly labeled, so that the canonical square on it is a pushout.

Definition 17 (Rewrite step). Let p be a rewrite rule ($L \xleftarrow{l} K \xrightarrow{r} R$) and $m : L \rightarrow G$ a matching, both consistent with a disconnection kit k of L . Then G rewrites to H using rule p if there is a diagram:

$$\begin{array}{ccccc}
 L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 m \downarrow & & \downarrow d & & \downarrow m' \\
 G & \xleftarrow{l'} & D & \xrightarrow{r'} & H
 \end{array}$$

where the left hand side of the diagram is the disconnection of m with respect to k and the right hand side is a canonical square.

So, according to theorems 3 and 4, a rewrite step corresponds to a *double pushout* in the category of graphs.

Proposition 1 (A description of the nodes). *With the notations and assumptions of definition 17, the representatives of the equivalence classes of nodes of $\mathcal{N}_R + \mathcal{N}_D$ can be chosen in such a way that: $\mathcal{N}_H^\Omega = (\mathcal{N}_G^\Omega - m(\mathcal{N}_L^\Omega)) + \mathcal{N}_R^\Omega$ and $\mathcal{N}_H^X = \mathcal{N}_G^X + (\mathcal{N}_R^X - r(\mathcal{N}_L^X))$ and $\mathcal{N}_H^R = r'(\mathcal{N}_D^R) \cup m'(\mathcal{N}_R^R)$.*

4.3 Graph Rewriting with Garbage removal

In this section we give a direct application of garbage removal via a left adjoint in rooted graph rewriting. Indeed, left adjoints preserve pushouts. Therefore it is possible to apply functor ∇ on the double push out. Then the composition of $\Delta_R \circ \Lambda'$ gives us the garbage free reduced graph. This schema applies to every double pushout settings, we illustrate this on a particular one.

Definition 18 (Rewrite step with garbage removal). Let p be a rewrite rule ($L \xleftarrow{l} K \xrightarrow{r} R$) and $m : L \rightarrow G$ a matching, both consistent with a disconnection kit k of L . Then G rewrites with garbage removal to P using rule p if there is a diagram:

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ m \downarrow & & \downarrow d & & \downarrow m' \\ G & \xleftarrow{l'} & D & \xrightarrow{r'} & H \end{array}$$

where the left hand side is the disconnection of m with respect to k and the right hand side is a canonical square, and $P = V(\Delta_R(\Lambda'(\nabla(H)))) = V(\Lambda(H))$.

Example 6. First, let us simulate a term rewrite rule. Consider the rule $f(x) \rightarrow g(b)$. In our setting it can be implemented by the following span s :

$$\begin{array}{ccc} \boxed{\begin{array}{c} n : f \\ \downarrow \\ m : \bullet \end{array}} & \xleftarrow{l} & \boxed{\begin{array}{cc} n : f & n[0] : \bullet \\ \downarrow & \\ m : \bullet & \end{array}} & \xrightarrow{r} & \boxed{\begin{array}{cc} n : f & o : g \\ \downarrow & \downarrow \\ m : \bullet & p : b \end{array}} \end{array}$$

Where l, r are the expected graph homomorphisms with $l(n[0]) = n$ and $r(n[0]) = o$. Then $G_{\mathfrak{G}}$ rewrites to $P_{\mathfrak{G}}$ by using the rule s

$$G_{\mathfrak{G}} = \underline{q : h} \begin{array}{c} \xrightarrow{\quad} \\ \xleftarrow{\quad} \end{array} \begin{array}{c} n : f \\ \xrightarrow{\quad} \\ \xleftarrow{\quad} \end{array} m : a \qquad P_{\mathfrak{G}} = p : b \leftarrow o : g \begin{array}{c} \xleftarrow{\quad} \\ \xrightarrow{\quad} \end{array} \underline{q : h}$$

Indeed one has the following double pushout in \mathbf{Gr} :

$$\begin{array}{ccccc} \boxed{\begin{array}{c} n : f \\ \downarrow \\ m : \bullet \end{array}} & \xleftarrow{l} & \boxed{\begin{array}{cc} n : f & n[0] : \bullet \\ \downarrow & \\ m : \bullet & \end{array}} & \xrightarrow{r} & \boxed{\begin{array}{cc} n : f & o : g \\ \downarrow & \downarrow \\ m : \bullet & p : b \end{array}} \\ \downarrow m & & \downarrow d & & \downarrow m' \\ \boxed{\begin{array}{c} \underline{q : h} \xrightarrow{\quad} n : f \\ \quad \quad \downarrow \\ \quad \quad m : a \end{array}} & \xleftarrow{l'} & \boxed{\begin{array}{cc} \underline{q : h} & n : f \\ \downarrow \downarrow & \downarrow \\ n[0] : \bullet & m : a \end{array}} & \xrightarrow{r'} & \boxed{\begin{array}{ccc} o : g & \xleftarrow{\quad} \underline{q : h} & n : f \\ \downarrow & \quad \quad \downarrow & \downarrow \\ p : b & \quad \quad m : a \end{array}} \end{array}$$

In this example $f(a)$ becomes unreachable because of edge redirection. Let $H_{[6]}$ be the graph at the bottom right of this double pushout. Then, $\Lambda(H_{[6]})$ is the marked graph $P_{[6]}$, as above.

The following graph illustrates well the role played by roots in garbage removal. Consider $G'_{[6]}$ where the root is now n , it rewrites to $P'_{[6]}$ with:

$$G'_{[6]} = \begin{array}{c} q : h \rightleftarrows n : f \\ \swarrow \quad \downarrow \\ m : a \end{array} \qquad P'_{[6]} = \begin{array}{c} o : g \\ \downarrow \\ p : b \end{array}$$

This simple example is not possible to simulate in [6] where garbage cannot be removed.

Another interesting property of graph rewriting with garbage removal is the management of roots. New roots can be introduced by simple rules like:

$$\begin{array}{c} n : f \\ \downarrow \\ m : a \end{array} \xleftarrow{l} \begin{array}{c} n : f \quad n[0] : \bullet \\ \downarrow \\ m : a \end{array} \xrightarrow{r} \begin{array}{c} n : f \quad \underline{o : r} \\ \downarrow \quad \swarrow \\ m : a \end{array}$$

Where $r(n[0]) = o$ and $l(n[0]) = n$. This rule adds a new root $\underline{o : r}$.

Dually, the number of roots can be reduced, in special circumstances: this can be done by associating two roots equally labeled. For instance consider the span:

$$\begin{array}{c} \underline{o1 : f} \quad \underline{o2 : f} \\ \downarrow \quad \swarrow \\ m : a \end{array} \xleftarrow{l} \begin{array}{c} \underline{o1 : f} \quad \underline{o2 : f} \\ \downarrow \quad \swarrow \\ m : a \end{array} \xrightarrow{r} \begin{array}{c} \underline{o : f} \\ \downarrow \\ m : a \end{array}$$

where $r(o1) = r(o2) = o$. Note that by injectivity hypothesis of matching on labeled nodes, the left hand side of the span must match two different roots. Note also that injectivity hypothesis on morphism r only applies to unlabeled nodes, thus $o1, o2$ can be collapsed to a single node o .

Example 7. Let us now consider a more complicated example: the in-place reversal of a list between two particular cells. For example, given the graph:

$$h \rightarrow rev \rightarrow \boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \boxed{4} \rightarrow \boxed{5} \dots$$

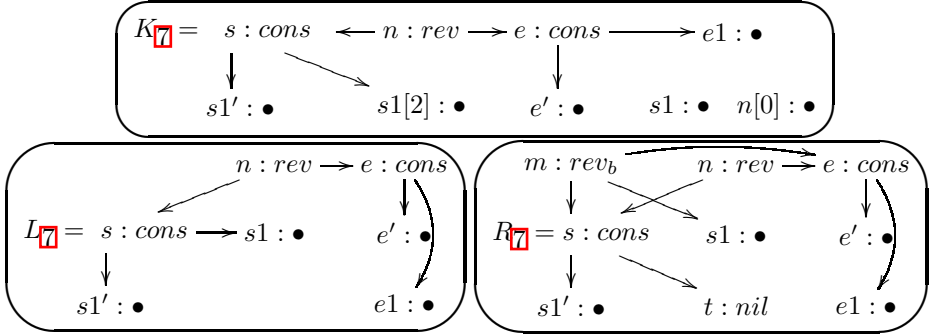
we want to produce the following graph: $nil \leftarrow \boxed{1} \leftarrow \boxed{2} \leftarrow \boxed{3} \leftarrow h$

Potentially, the rest of the graph should be removed. rev is defined by means of four rules. Moreover, one can notice that the programmer does not need to take a particular care of garbage management: it is automatically managed. The first rule is for trivial cases (when the first and last items are equal) and is implemented as follows:

$$\begin{array}{c} n : rev \rightleftarrows m : \bullet \\ \rightleftarrows \end{array} \xleftarrow{l} \begin{array}{c} n : rev \rightleftarrows m : \bullet \\ \rightleftarrows \\ n[0] : \bullet \end{array} \xrightarrow{r} \begin{array}{c} n : rev \rightleftarrows m : \bullet \\ \rightleftarrows \end{array}$$

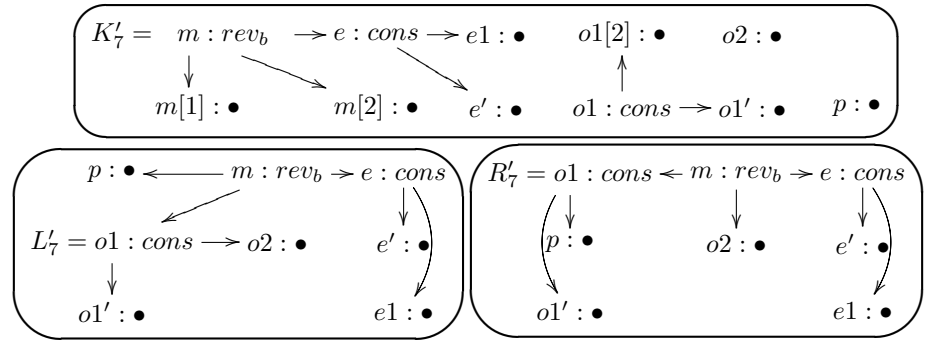
This rule only performs a global redirection from n to m ($r(n[0]) = m$ and $l(n[0]) = n$), thus node $n : rev$ will be garbage removed after the rewrite step (nothing can no longer points to it because of the global redirection).

The second rule classically introduces an auxiliary function rev_b of three parameters which performs the actual rewriting of the list. The first two parameters of rev_b record the pair of list cells to be inverted (current and preceding cells) and the last parameter stores the halting cell. It is done by the span $L_7 \leftarrow K_7 \rightarrow R_7$ where:



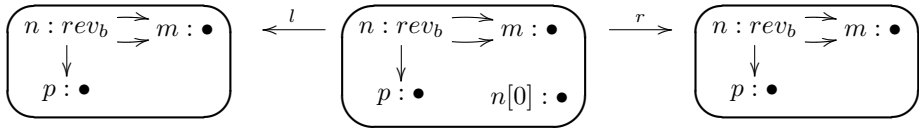
where $n[0] : \bullet$ mapped to n on L_7 and to m on R_7 , $s1[2]$ is mapped to t in R_7 .

The general step of rev_b is given by the span $L'_7 \leftarrow K'_7 \rightarrow R'_7$, where:



where $m[1], m[2]$ are respectively mapped to $o1, o2$ and $o1[2]$ is mapped to p in R'_7 . This rule disconnects the first two parameters of rev_b ($m[1], m[2]$) to make them progress along the list (p is replaced by $o1$ and $o1$ by $o2$). It also redirects the local edge of the list to be reversed (second edge from $o1$) to the previous cell of the list (node p). One has to remember the injectivity hypothesis of the matching homomorphism on labeled nodes. It ensures that nodes $o1, e$ are actually different nodes. Thus there can be no confusion with the halt case.

The halt case for rev_b is similar to the one on rev and just amounts to a global redirection, namely:



where $n[0]$ is mapped to m on the graph on the right side.

We let the reader check on examples that these rules implement the in-situ list reversal between two given nodes.

5 Conclusion

We have presented a categorical approach to garbage collection and garbage removal which can be applied to various graph rewriting frameworks, especially the ones based on pushouts. Garbage removal may be seen either as a right adjoint or as a left adjoint. The right adjoint is the mathematical translation of the description of what is garbage collection: the removal of unreachable parts. On the other hand the left adjoint gives an operational point of view. It illustrates well the three basic steps of any real garbage collector in programming languages: when a garbage collector starts there is first a propagation phase to compute the live parts, then follows the removal of non live nodes and finally the halt of the garbage collector (the return to a normal evaluation mode). Those three steps are represented by three associated functors.

As a future work, we plan to use graph transformation frameworks in order to model memory, as well as mutable objects, transformation.

References

1. Banach, R.: Term graph rewriting and garbage collection using opfibrations. *Theoretical Computer Science* 131, 29–94 (1994)
2. Barendregt, H., van Eekelen, M., Glauert, J., Kenneway, R., Plasmeijer, M.J., Sleep, M.: Term graph rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) *PARLE'87*. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
3. Broek, P.V.D.: Algebraic graph rewriting using a single pushout. In: Abramsky, S. (ed.) *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development*. LNCS, vol. 493, pp. 90–102. Springer, Heidelberg (1991)
4. Cohen, J.: Garbage collection of linked data structures. *Computing Surveys* 13(3), 341–367 (1981)
5. Collins, G.: A method for overlapping and erasure of lists. *Communication of the ACM* 3(12), 655–657 (1960)
6. Duval, D., Echahed, R., Prost, F.: Modeling pointer redirection as cyclic term graph rewriting. In: *TERMGRAPH 06 (2006)* (Extended version to appear in *ENTCS*)
7. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: *FOCS 1973*, pp. 167–180 (1973)
8. Jones, R.E., Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. J. Wiley & Son, New York (1996)
9. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine-i. *Communication of the ACM* 3(1), 184–195 (1960)

Non Strict Confluent Rewrite Systems for Data-Structures with Pointers^{*}

Rachid Echahed and Nicolas Peltier

LIG, CNRS

46, avenue Félix Viallet

38031 Grenoble Cedex, France

Rachid.Echahed@imag.fr, Nicolas.Peltier@imag.fr

Abstract. We introduce a notion of rewrite rules operating on a particular class of data-structures, represented as (cyclic) term-graphs. All basic transformations are available: node creation/deletion, node relabeling and edge redirections (including global redirections). This allows one to write algorithms handling pointers that cannot be efficiently specified using existing declarative languages. Such rewrite systems are not confluent in general, even if we stick to orthogonal, left-linear rules. In order to ensure unique normal forms, we introduce a notion of *priority ordering* between the nodes, which allows the programmer to control the normalization of a graph if needed. The use of total priority orderings makes rewriting purely deterministic, which is not always efficient in practice. To overcome this issue, we then show how to define more flexible strategies, which yield shorter derivations and avoid useless rewriting steps (lazy rewriting).

1 Introduction

Systems of rewrite rules [7] provide a very natural and convenient way of specifying algorithms. Assume for instance that we want to define a function *add_last* adding an element x at the end of a list l . This can be done using the following rules:

$$\text{add_last}(x, \text{nil}) \rightarrow \text{cons}(x, \text{nil}) \quad (\rho_1)$$

$$\text{add_last}(x, \text{cons}(y, l)) \rightarrow \text{cons}(y, \text{add_last}(x, l)) \quad (\rho_2)$$

However, these rules have an obvious drawback from a programming point of view: they do not really *insert* x at the end of l (as it could be done in any imperative language, using pointer redirections), but rather *reconstruct* the list entirely. Additional memory is consumed, which could be in principle avoided. Even if the memory can be freed afterwards using garbage collection (which is not always the case), additional computation time is consumed.

Thus it would be very desirable to allow the programmer to specify how the memory should be used and which “nodes” should be reused. Many pointer-based algorithms extensively rely on this possibility, for instance, the Schorr-Waite algorithm [14] uses a link reversal technique to avoid the need for a stack

^{*} This work has been partly funded by the project ARROWS of the French *Agence Nationale de la Recherche*.

during the exploration of a graph. In-situ algorithms for sorting or reversing a list also require such capabilities.

Thus we propose to consider *more expressive rewrite rules*, able, not only to create new terms, but also to *physically modify* a term by *redirecting* some of the “edges” occurring in it. For instance, the rule ρ_2 can be replaced by a rule (ρ'_2 below) redirecting the tail of the list $cons(y, l)$ to the list $add_last(x, l)$. The expression $add_last(x, l)$ is obtained itself by redirecting the second argument of add_last in $add_last(x, cons(y, l))$ to l . Then *no new term is created* (except for the last element). This rule can be written as follows (the formal definition of the graphs and rules will be given in Sections 2 and 3).

$$\beta : add_last(x, \alpha : cons(y, l)) \rightarrow \beta \gg_2 l \quad (\rho'_2).$$

$\beta \gg_2 l$ denotes an action which redirects the second argument of β to l . The node β is reused in the right-hand side in order to avoid creating new cells. Note that such rewrite rules are expressive enough to create shared or cyclic data-structures. For instance, the action $\alpha \gg_2 \alpha$ creates a cyclic list $\alpha : cons(y, \alpha)$ of length 1.

In this paper, we investigate a class of rewrite rules having the following properties. First we handle data-structures that are more complex than standard terms, corresponding to a particular class of graphs (see Section 2). Note that this is not really important by itself, since such data-structures could be in principle encoded into terms (possibly using built-in functions). This should be considered in connection with the second point. Second, we introduce a language able to express all basic transformations on such data-structures, including node relabeling, edge (pointer) redirection and global redirection¹. The right-hand sides of the rules we consider in this paper are defined as sequences of elementary actions. These actions, such as (re)definition of nodes, global or local redirection of edges/pointers, contribute to rewrite a graph stepwise. These systems are thus different from classical term graph rewriting systems such as [12, 4].

In [3] graph grammars operating on data-structures with pointers have been proposed as a means to recognize shapes of data-structures with pointers. That is to say, given a graph G representing a data structure, one may use a graph grammar to solve the word problem by answering whether or not G belongs to a set of graphs having a common shape. The class of graph rewrite rules we investigate in the present paper can also be used to recognize shapes of data-structures. However, we will rather focus on the computational aspects related to the proposed rewrite system instead of focusing on some applications such as shape recognition.

Our language is a *conservative extension* of term rewrite rules. However, since our rewrite rules are more “expressive” than usual ones, confluence is obviously much more difficult to ensure. Assume for instance that we want to normalize the following term-graph: (α, β) where α, β are defined as follows: $\alpha : add_last(0, \delta)$, $\beta : add_last(1, \delta)$, where $\delta : cons(2, nil)$. Then the normal form of the graph depends

¹ This last transformation consists in redirecting all the edges pointing to a given node to another node in the graph. It is needed to express collapsing rules, for instance $cdr(cons(x, l)) \rightarrow l$: any edge pointing to $cdr(cons(x, l))$ should be redirected to l .

on the order in which the nodes are reduced. The resulting list is either $cons(2, cons(0, cons(1, nil)))$ (if α is reduced first), or $cons(2, cons(1, cons(0, nil)))$ (if β is reduced first).

The property of confluence is essential when rewrite rules are used to define functions. It ensures the uniqueness of normal forms and allows for more efficient rewrite strategies (since one does not have to explore all possible derivations). Unfortunately, this property turns out to be hard to satisfy for graph rewrite systems see e.g., [13,2,11]. In order to overcome this problem, we propose to associate each graph to a **priority ordering** (chosen by the programmer) between the (non constructor) nodes, expressing the order in which the nodes should be reduced – thus enforcing confluence. However, it is not always necessary to order all the nodes. If, for instance, the function add_last is applied to two lists l, l' sharing no nodes, then it is clear that the two corresponding terms can be evaluated in any order, since their executions do not interfere. If standard rewrite rules are considered then arbitrary rewrite strategies should be possible (all nodes can be reduced as soon as possible). This raises the following issue: How can we decide which nodes should be ordered ?

In [5] we made a first attempt to ensure unique normal forms by ordering any two reducible positions which may produce side-effects (e.g., edge redirections). However, these orderings are too restrictive and prevent in general to perform efficient evaluation strategies. We propose here more refined orderings over nodes which allow us to retrieve the efficiency of lazy strategies whenever it is possible such as the optimal strategy defined in [8]. These orderings take into account not only the functions labeling the nodes, but also the *dependencies* between the nodes. These priorities over nodes are characterized in an abstract way, we call *dependency schemata*. Several orderings can fulfill such schemata. We provide in addition an actual procedure to define such refined orderings. To our knowledge, there is no similar result in the literature which computes dynamically a set of positions which may be candidate to be reducible without loosing confluence. The computation of such a set of positions is different from expressing strategies as in [15,6] or computing needed positions as in [10].

Before giving any technical definition, we would like to provide an **informal overview** of our approach (the reader can refer to Section 2.1 for the notations).

Strict Rewriting. First, we assume given a *total* ordering, denoted by \succeq , on the nodes (Section 3). Since the ordering is total, the corresponding derivations are *purely deterministic* (if the rewrite rules are orthogonal). Thus the normal forms are unique. This is useful for defining the *semantics* of a set of rules, but this is not satisfactory from a programming point of view, because it prevents us from using efficient, lazy, rewriting strategies, or parallelism.

For instance, given a rule $\alpha:(\beta:0 \times \gamma) \rightarrow \alpha:0$, and a term $t = (\beta:0) \times s$, where the nodes in s are of highest priority than β (and not connected to β), one would have to evaluate the term s before finding the value of t , although obviously the value of t is always 0. Clearly, this is inefficient, since s may be arbitrarily complex. Even worse, the evaluation of s may not terminate which entails that t itself may be non normalizable. We have here a similar behavior

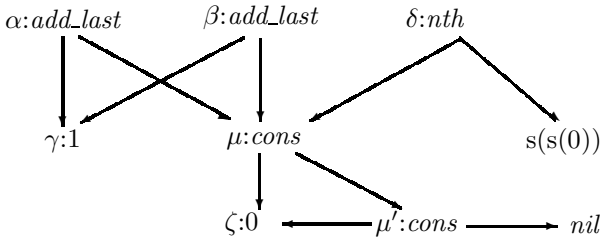
of innermost term rewriting w.r.t. lazy (outermost) term rewriting. Thus, more flexible rewrite strategies are needed.

Non Strict Rewriting. We define relaxed, flexible rewrite strategies with the following properties: confluence is preserved, which implies that the computed normal forms are the same as the ones computed by strict rewriting (i.e. the *semantics* of the rules, as defined by strict rewriting, is preserved), but at the same time, *shorter derivations can be obtained*, by skipping some useless rewrite steps.

The general idea is the following. We allow to apply a given rule ρ on a node β , even if it is not the one with highest priority, if the two following properties can be ensured: (i) The left-hand side of ρ does not contain any node that is affected (i.e. redirected or relabeled) during the normalization of the nodes α s.t. $\alpha \succeq \beta$; **and** (ii) ρ does not affect a node which is “reachable”² from a node α s.t. $\alpha \succeq \beta$. However, it is not always easy to check whether the normalization of a given subgraph affects or not another subpart of the term-graph. For example, in the example above, the evaluation of s may affect α : assume for instance that $s = g(\beta)$, where g is defined by the rule $\alpha:g(\beta:0) \rightarrow \alpha:2, \beta:1$ (g returns 2 and changes the label of its argument to 1). Then $t = (\beta:0) \times g(\beta)$ may be rewritten to (1×2) .

These properties are undecidable in general, thus we approximate them using *tractable* criteria (Section 4).

An Illustrating Example. We give an example illustrating the differences between the two kinds of rewriting. This example is intended to give an intuition of what we want to achieve, and to demonstrate the interest of the strategy we introduce in the paper.



Assume that we want to normalize the above term-graph, where the priority ordering is defined as follows: $\alpha \succeq \beta \succeq \delta$. We want to compute the value of δ .

This term-graph can be described linearly as follows (see Section 2.1 for details): $\alpha: add_last(\gamma:1, \mu: cons(\zeta:0, \mu': cons(\zeta, \eta: nil))) \oplus \beta: add_last(\gamma, \mu) \oplus \delta: nth(\mu, s(s(0)))$. The function nth (n th element in a list) is defined as usual by the following rules π_1 and π_2 .

$$nth(cons(x, l), s(0)) \rightarrow x \quad (\pi_1) \quad nth(cons(x, l), s(s(n))) \rightarrow nth(l, s(n)) \quad (\pi_2)$$

Strict rewriting would first reduce the node α yielding eventually:

² In a sense to be specified: it is not sufficient to explore the current graph, because the node may become reachable after some rule application.

$$\beta: \text{add_last}(\gamma:1, \mu: \text{cons}(\zeta:0, \text{cons}(\zeta, \text{cons}(\gamma, \eta: \text{nil})))) \oplus \delta: \text{nth}(\mu, s(s(0))).$$

Then the node β will be reduced, which produces:

$$\delta: \text{nth}(\mu: \text{cons}(\zeta:0, \mu': \text{cons}(\zeta, \text{cons}(\gamma:1, \text{cons}(\gamma, \eta: \text{nil}))))), s(s(0))).$$

Afterwards, the function nth will be applied, yielding $\zeta:0$. The rewrite strategy described in [5] would yield the same derivation.

Now, let us see what happens when applying the more flexible strategy developed in this paper. Initially, we still have to rewrite α . Indeed, it is the node with highest priority. Moreover, it may affect the node μ , which is reachable from δ . Note that the rule π_2 could be applied in principle on δ (it matches the graph at node δ), but is *blocked* at this point by the ordering, indeed, it applies to a part of the graph that contains a node μ that is reachable from two nodes α, β which may perform “side-effects” (i.e. may physically affect their arguments). Thus rewriting δ at this point is “unsound” (w.r.t. the semantics defined by the ordering \succeq on nodes) because μ could be relabeled or redirected when normalizing α or β . Thus we apply the rule ρ'_2 on α . The term-graph becomes:

$$\alpha: \text{add_last}(\gamma:1, \mu': \text{cons}(\zeta:0, \eta: \text{nil})) \oplus \beta: \text{add_last}(\gamma, \mu: \text{cons}(\zeta, \mu')) \oplus \delta: \text{nth}(\mu, s(s(0))).$$

However, **we do not need to rewrite α again**. Indeed, the rule ρ'_2 is applicable on β and the part of the graph on which it applies is not affected by α (formal criteria will be given later). Moreover, this rule does not affect the nodes reachable from α . Thus conditions (i) and (ii) are satisfied and we can *delay* the reduction of α (lazy rewriting) and apply instead ρ'_2 on β (even if $\alpha \succeq \beta$). We obtain: $\alpha: \text{add_last}(\gamma:1, \mu': \text{cons}(\zeta:0, \eta: \text{nil})) \oplus \beta: \text{add_last}(\gamma, \mu') \oplus \delta: \text{nth}(\mu: \text{cons}(\zeta, \mu'), s(s(0)))$.

At this point, we can apply the rule π_2 . Indeed, it affects no node (no side-effect) and it applies on a part of the graph that is not reachable from α or β any more. This yields: $\alpha: \text{add_last}(\gamma:1, \mu': \text{cons}(\zeta:0, \eta: \text{nil})) \oplus \beta: \text{add_last}(\gamma, \mu') \oplus \delta: \text{nth}(\mu', s(0))$.

Now, we have no other choice than applying ρ'_2 on α (this is the only applicable rule) and get: $\alpha: \text{add_last}(\gamma:1, \eta: \text{nil}) \oplus \beta: \text{add_last}(\gamma, \mu': \text{cons}(\zeta:0, \eta)) \oplus \delta: \text{nth}(\mu', s(0))$. Then, again, we can delay the reduction of α and reduce β instead. This yields: $\alpha: \text{add_last}(\gamma:1, \eta: \text{nil}) \oplus \beta: \text{add_last}(\gamma, \eta) \oplus \delta: \text{nth}(\mu': \text{cons}(\zeta:0, \eta), s(0))$.

At this point, we do not need to reduce α or β . We can directly apply the rule π_1 on δ , which gives the result 0. Afterwards, the remaining nodes can be deleted (see Section [5]).

Due to space restrictions, the proofs are not included.

2 Basic Definitions

2.1 Term-Graphs

The class of term-graphs³ we consider in this paper is slightly different from the one of our previous papers [5],[9]. It is close to the formalism of Ψ -terms [1].

³ We use the word “term-graph” in this paper in order to emphasize the difference between the considered data-structures and the graphs usually used in mathematics: mainly, given a node α and a symbol a , there can be at most one edge starting from α and labeled by a .

We assume given a set of *nodes* \mathcal{N} , a set of *features* \mathcal{F} and a set of *labels* \mathcal{L} . In contrast to [5,9], our signature is *constructor-based*: \mathcal{L} is divided into two disjoint sets of symbols: a set \mathcal{D} of *defined symbols* and a set \mathcal{C} of *constructors*.

Nodes will be denoted by Greek letters, labels by f, g, \dots and features by a, b, \dots . Features may be seen as functions from \mathcal{N} to \mathcal{N} , or as edge labels.

We assume given an ordering \succeq on \mathcal{N} , called the *priority ordering*. It expresses in which order the nodes should be reduced. Note that in contrast to [5], \succeq is assumed to be total.

Definition 1. *A term-graph t is defined by:*

- A set of nodes $\mathcal{N}(t) \subseteq \mathcal{N}$.
- A partial labeling function l_t from $\mathcal{N}(t)$ to \mathcal{L} . $l_t(\alpha)$ denotes a symbol labeling the node α .
- A function mapping each symbol a in \mathcal{F} to a partial function a_t from $\mathcal{N}(t)$ to $\mathcal{N}(t)$. If $a_t(\alpha) = \beta$ then we say that t contains an edge from α to β , labeled by a .

A rooted term-graph is a term-graph associated with a distinguished node α , called the root of t and denoted by $root(t)$.

Let t, s be two term-graphs. We write $t \subseteq s$ iff t is included in s i.e. iff $\mathcal{N}(t) \subseteq \mathcal{N}(s)$ and the functions l_t and a_t (for any $a \in \mathcal{F}$) are restrictions of l_s and a_s respectively.

A Linear Notation for Term-Graphs. We introduce a linear notation for denoting term-graphs, which is close to the one used for terms (and Ψ -terms [1]) and more convenient to use than the above definition.

If t_1, \dots, t_n are rooted term-graphs, we denote by $\alpha: f(a_1 \Rightarrow t_1, \dots, a_n \Rightarrow t_n)$ the minimal term-graph t (if it exists) containing the term-graphs t_1, \dots, t_n , the node α and such that $l_t(\alpha) = f$, and for all $i \in [1..n]$, $(a_i)_t(\alpha) = root(t_i)$ ⁴. $t \oplus s$ denotes the union of t and s (if it exists).

α can be left unspecified, in the case it is simply replaced by an arbitrary node not occurring elsewhere. $f(t_1, \dots, t_n)$ is syntactic sugar for $f(1 \Rightarrow t_1, \dots, n \Rightarrow t_n)$.

A difference with the definition used, e. g., in [5,9] is that we do not need to define simultaneously all the “arguments” (i.e. the features) of a given node. For instance, $f(2 \Rightarrow b)$ denotes a term in which the second argument is b and the first one is undefined.

Additional conditions can be added in order to ensure that the term-graphs are well-formed (for instance that the arities of the function symbols are respected). We also assume in the paper that the rewrite rules preserve well-formedness. This problem is obviously undecidable in general, thus appropriate syntactic restrictions should be added on the rewrite rules (for instance, if we do not want to consider undefined features, then only the nodes that are globally redirected

⁴ Note that t does not necessarily exists since some of the nodes may be redefined. For instance, $\alpha: f(a \Rightarrow \beta:0, b \Rightarrow \beta:1)$ contains two contradictory labelings of the node β .

should be deleted). We do not consider this issue in the present paper, because the notion of well-formedness strongly depends on the considered application.

2.2 \mathcal{N} -Mappings

An \mathcal{N} -mapping is a *total* function from \mathcal{N} to \mathcal{N} . An \mathcal{N} -mapping σ is said to be *compatible* with a term-graph t if for all $\alpha, \beta \in \mathcal{N}(t)$ s.t. $\sigma(\alpha) = \sigma(\beta)$, the two following conditions hold:

- If $l_t(\alpha)$ and $l_t(\beta)$ are defined then we have $l_t(\alpha) = l_t(\beta)$.
- for all $a \in \mathcal{F}$ s.t. $a_t(\alpha)$ and $a_t(\beta)$ are defined, then $\sigma(a_t(\alpha)) = \sigma(a_t(\beta))$.

Then $\sigma(t)$ denotes the term-graph s defined as follows: $\mathcal{N}(s) \stackrel{\text{def}}{=} \{\sigma(\alpha) \mid \alpha \in \mathcal{N}(t)\}$, $l_s(\sigma(\alpha)) \stackrel{\text{def}}{=} l_t(\alpha)$ and $a_s(\sigma(\alpha)) \stackrel{\text{def}}{=} \sigma(a_t(\alpha))$.

An \mathcal{N} -mapping σ is said to be a *renaming* for a term-graph t if σ is injective and if for any pair of node α, β occurring in t , $\alpha \succeq \beta \Rightarrow \sigma(\alpha) \succeq \sigma(\beta)$. Note that by definition any renaming for t is compatible with t .

An \mathcal{N} -relation Δ is a function mapping any term-graph t to a relation Δ_t on the nodes in t s.t. for any renaming η and for any pair of nodes α, β occurring in t we have $\alpha \Delta_t \beta$ iff $\eta(\alpha) \Delta_{\eta(t)} \eta(\beta)$.

One of the simplest examples of an \mathcal{N} -relation is the relation \rightarrow_t defined as the smallest reflexive and transitive relation s.t. $(\alpha \in \mathcal{N}(t) \wedge a_t(\alpha) = \beta) \Rightarrow \alpha \rightarrow_t \beta$ (informally $\alpha \rightarrow_t \beta$ means that there is a path from α to β in t).

2.3 Actions

The definitions of actions and rewriting rules are close to the ones of [9]. An *action* is one of the following forms:

- a **node relabeling** $\alpha:f$ where α is a node and f is a label. This means that α is (re)labeled by f .
- an **edge redirection** $\alpha \gg_a \beta$ where α, β are nodes and a is a feature. This means that the value of $a(\alpha)$ is changed to β . This may be seen as an edge redirection: the target of the edge starting from α and labeled by a is redirected to point to β . The edge is created if it does not exist.
- a **global redirection** $\alpha \gg \beta$ where α and β are nodes. This means that all edges pointing to α are redirected to β .
- a **node deletion** $\bar{\alpha}$ where α is a node.
- an **edge deletion** $\bar{\alpha}_a$ where α is a node and a is a feature.

The result of applying an action ϵ to a term-graph t is denoted by $\epsilon[t]$ and is defined as the following term-graph s :

- If $\epsilon = \alpha:f$ then $\mathcal{N}(s) \stackrel{\text{def}}{=} \mathcal{N}(t) \cup \{\alpha\}$, $l_s(\alpha) \stackrel{\text{def}}{=} f$, $l_s(\beta) \stackrel{\text{def}}{=} l_t(\beta)$ if $\beta \neq \alpha$, and for any feature a , $a_s \stackrel{\text{def}}{=} a_t$. \cup denotes classical union.
- If $\epsilon = \alpha \gg_a \beta$ then $\mathcal{N}(s) \stackrel{\text{def}}{=} \mathcal{N}(t) \cup \{\alpha, \beta\}$, $l_s \stackrel{\text{def}}{=} l_t$, $a_s(\alpha) \stackrel{\text{def}}{=} \beta$ and for any feature b and any node γ we have $b_s(\gamma) \stackrel{\text{def}}{=} b_t(\gamma)$ iff $a \neq b$ or $\gamma \neq \alpha$.

- If $\epsilon = \alpha \gg \beta$, then $\mathcal{N}(s) \stackrel{\text{def}}{=} \mathcal{N}(t) \cup \{\alpha, \beta\}$, $l_s \stackrel{\text{def}}{=} l_t$, and for any feature a and any node γ , $a_s(\gamma) \stackrel{\text{def}}{=} \beta$ if $a_t(\gamma) = \alpha$ and $a_s(\gamma) \stackrel{\text{def}}{=} a_t(\gamma)$ otherwise.
- If $\epsilon = \bar{\alpha}$, then $\mathcal{N}(s) \stackrel{\text{def}}{=} \mathcal{N}(t) \setminus \{\alpha\}$, and for any node $\beta \neq \alpha$, $l_s(\beta) \stackrel{\text{def}}{=} l_t(\beta)$ and for any feature a , $a_s(\beta) \stackrel{\text{def}}{=} a_t(\beta)$ if $a_t(\beta) \neq \alpha$ ($a_s(\beta)$ is undefined otherwise).
- If $\epsilon = \bar{\alpha}_a$ then $\mathcal{N}(s) \stackrel{\text{def}}{=} \mathcal{N}(t) \cup \{\alpha\}$, $l_s \stackrel{\text{def}}{=} l_t$ and for any feature b and any node γ we have $b_s(\gamma) \stackrel{\text{def}}{=} b_t(\gamma)$ iff $a \neq b$ or $\gamma \neq \alpha$.

$\zeta[t]$ is defined inductively as follows: $\zeta[t] \stackrel{\text{def}}{=} t$ if ζ is empty and $(\epsilon.\zeta)[t] \stackrel{\text{def}}{=} \zeta[\epsilon[t]]$ (where ϵ is an action and $.$ denotes the sequential operator).

Example 1. Let $t = \alpha:f(\beta:a,\gamma:g(\beta),\alpha)$. We have:

$(\alpha:h.\gamma:f)[t] = \alpha:h(\beta:a,\gamma:f(\beta),\alpha)$, $(\alpha \gg_1 \gamma)[t] = \alpha:f(\gamma:g(\beta:a),\gamma,\alpha)$ and $(\beta \gg \gamma)[t] = \alpha:f(\gamma:g(\gamma),\gamma,\alpha)$.

3 Term-Graph Rewriting

Definition 2. A node constraint is a (possibly empty) conjunction of disequations between nodes: $\bigwedge_{i=1}^n (\alpha_i \neq \beta_i)$. An \mathcal{N} -mapping σ is a solution of a constraint $\phi = \bigwedge_{i=1}^n (\alpha_i \neq \beta_i)$ iff for any $i \in [1..n]$, we have $\sigma(\alpha_i) \neq \sigma(\beta_i)$. We denote by $\text{sol}(\phi)$ the set of solutions of ϕ .

Definition 3. (Rewrite Rule) A (constrained) term-graph rewrite rule is an expression of the form $[L \rightarrow R \mid \phi]$ where R is a sequence of actions, ϕ is a constraint and L is a rooted term-graph s.t.:

- for any node α occurring in L , we have $\text{root}(L) \rightarrow_L \alpha$ (i.e. any node occurring in the left-hand side must be reachable from the root⁵).
- R contains at most one global redirection, which is of the form $\text{root}(L) \gg \beta$ (only the root may be globally redirected).
- We have $l_L(\text{root}(L)) \in \mathcal{D}$ and for any node $\alpha \neq \text{root}(L)$ if $l_L(\alpha)$ is defined then $l_L(\alpha) \in \mathcal{C}$ (the root is the only node in L that is labeled by a defined symbol).

A rewrite system is a set of rewrite rules.

Example 2. The following rules physically reverse a list (i.e. without creating new cells). A list is represented by a term labeled by *cons* or *nil*, with two features *car* (current element) and *cdr* (the tail).

The first rule replaces a term-graph $\text{rev}(\beta)$ by $\text{aux}(\beta, \text{nil})$, where *nil* is a new node. *aux* denotes an auxiliary function, which reverses its first argument and (physically) appends it to the second list.

$$\alpha:\text{rev}(\beta) \rightarrow \alpha:\text{aux} . \alpha \gg_2 \gamma . \gamma:\text{nil}.$$

⁵ This is an important condition since otherwise the same rule could be applied in several different ways at the *same* node.

The last two rules are applied to term-graphs of the form $\alpha:aux(\beta, \gamma)$. The second rule handles the case where β is *nil*. In this case, the result is γ , thus the node α is globally redirected to γ . Note that global redirections are essential for defining such rules. Afterwards, the node α can be deleted ($\bar{\alpha}$).

$$\alpha:aux(\beta:nil, \gamma) \rightarrow (\alpha \gg \gamma) . \bar{\alpha}.$$

The last rule (inductive case) handles the case where β is of the form *cons* ($car \Rightarrow \lambda, cdr \Rightarrow \delta$). Then the following actions are performed: the tail (*cdr*) of β is directed to γ , and the function *aux* is called on δ, β .

$$\alpha:aux(\beta:cons(car \Rightarrow \lambda, cdr \Rightarrow \delta), \gamma) \rightarrow (\beta \gg_{cdr} \gamma) . (\alpha \gg_1 \delta) . (\alpha \gg_2 \beta).$$

Definition 4. Let $\rho : [L \rightarrow R \mid \phi]$ be a rule. A ρ -matcher for a term-graph t (at a node $\alpha \in \mathcal{N}(t)$) is an \mathcal{N} -mapping σ compatible with L satisfying the following conditions.

1. σ is a solution of ϕ .
2. $\sigma(L) \subseteq t$.
3. $\alpha = \sigma(\text{root}(L))$.
4. Let N be the set of nodes occurring in R but not in L . σ maps the nodes in N to pairwise distinct nodes not occurring in t s.t.:
 - If β, γ are two nodes in N s.t. $\beta \succeq \gamma$ then $\sigma(\beta) \succeq \sigma(\gamma)$.
 - For any node β occurring in t , and for any node γ in N , $\beta \prec \sigma(\gamma)$ iff $\beta \prec \alpha$ or $\beta = \alpha$.

The first three conditions are natural. They express the fact that $\sigma(L)$ is a term-graph included in t , of root α , and that σ satisfies the constraint of the rule. The last condition states that the “extra” nodes, i.e. the nodes that occur in the right-hand side but not in the left-hand side should be mapped to new – pairwise distinct – nodes. This is very natural, since they correspond to nodes that are “created” by the rule application. The condition also specifies how those new nodes are ordered with respect to the nodes already existing in the term-graph: roughly speaking, those nodes “inherit” the priority of the parent node α .

If σ is a ρ -matcher for t at α then we denote by $\rho^\sigma[t]$ the term-graph s s.t. $s = \sigma(R)[t]$. Obviously, ρ -matchers can be easily computed using standard matching algorithms.

We write $t \xrightarrow{(\rho, \sigma)} s$ if $s = \rho^\sigma[t]$ and if the two following conditions are satisfied.

- For each node $\alpha \succ \sigma(\text{root}(L))$ occurring in t we have $l_t(\alpha) \in \mathcal{C}$. This means that the rules are applied only on maximal reducible nodes (according to the ordering \succeq).
- For any node α occurring in L , if $l_L(\alpha)$ is undefined then $l_t(\sigma(\alpha)) \notin \mathcal{D}$. This means that the variable nodes occurring in L cannot be labeled by a defined symbol [6](#).

⁶ This condition is essential for avoiding cycles in global redirections: for instance if the rules $\alpha:f(\beta) \rightarrow \alpha \gg \beta$ and $\alpha:g(\beta) \rightarrow \alpha \gg \beta$ are applied on $\gamma:f(g(\gamma))$. Without the above condition we could obtain two distinct normal forms $\gamma:f(\gamma)$ and $\gamma:g(\gamma)$.

These conditions specify the rewrite strategy. This strategy is very restrictive and it should be clear that it is not intended to be used in practice for normalizing term-graphs. It is useful only to specify the *semantics* of the rules.

We write $t \xrightarrow{\rho} s$ if $t \xrightarrow{(\rho,\sigma)} s$ for some ρ -matcher σ . We write $t \xrightarrow{\mathcal{R}} s$ if $t \xrightarrow{\rho} s$ for some rule $\rho \in \mathcal{R}$. This relation is called *strict rewriting*. One can view strict rewrite systems as purely “imperative” programs, in the sense that the order in which the actions are performed is entirely specified.

As explained in the introduction, we now introduce more flexible, non deterministic strategies that can be more efficient than strict rewriting, but in the same time compute the same normal forms (confluence is preserved).

The basic idea is that a rule ρ may be applied on a non-maximal node α only if ρ does not interfere with the reduction of the nodes $\beta \succ \alpha$, i.e. if the execution of ρ affects no node on which the functions labeling the nodes $\beta \succ \alpha$ operate, **and** if the left-hand side of ρ cannot be modified during the normalization of the nodes $\beta \succ \alpha$. Thus we need a way to determine (in a purely automatic and efficient way) which are the nodes that are affected by the normalization of a given node. However, this problem is clearly undecidable. Thus we need to formulate necessary, tractable, conditions.

We need to introduce some technical definitions. A node α is said to be *defined* in a term-graph t if either $l_t(\alpha)$ is defined (i.e. α has a label) or if there exists a feature a s.t. $a_t(\alpha)$ is defined (i.e. there is an edge starting from α). Non-defined nodes may be viewed as “variables”.

Let ς be a sequence of actions. We say that ς *redirects* α if ς contains an action of the form $\alpha \gg \beta$. We say that ς *affects* a node α if either it redirects α or if ς contains an action of the form $\alpha:f$ or $\alpha \gg_a \beta$ or $\bar{\alpha}$ or $\bar{\alpha}_a$. This means that the application of ς either changes the label of the node α , or redirects/deletes an edge starting from α or deletes α .

Let t be a term-graph. We define the following relations between nodes:

- $\alpha >_t \beta$ iff there is a rule $\rho : [L \rightarrow R \mid \phi] \in \mathcal{R}$ and a ρ -matcher σ at α s.t. β is defined in $\sigma(L)$. This means that a rule that can be applied on α depends on the definition of the node β .
- $\alpha \Rightarrow_t \beta$ iff there is a rule $\rho : [L \rightarrow R \mid \phi] \in \mathcal{R}$ and a ρ -matcher σ at α s.t. $\sigma(R)$ affects β . This means that a rule affecting β can be applied on α .
- $\alpha \rightsquigarrow_t \beta$ iff there is a rule $\rho : [L \rightarrow R \mid \phi] \in \mathcal{R}$ and a ρ -matcher σ at α s.t. $\sigma(R)$ redirects β . This means that a rule redirecting β can be applied on α .
- $\alpha \sqsupset_t \beta$ iff there is a rule $\rho : [L \rightarrow R \mid \phi] \in \mathcal{R}$ and a ρ -matcher σ at α s.t. β occurs in $\sigma(\phi)$. This means that a rule that can be applied on α depends on the *name* of β .⁷

Definition 5. Let \mathcal{R} be a set of rewriting rules. An \mathcal{R} -dependency schema is a triple $\xi = (\Rightarrow^\xi, \rightsquigarrow^\xi, >^\xi)$ of \mathcal{N} -relations, s.t. $\Rightarrow_t^\xi, \rightsquigarrow_t^\xi, >_t^\xi$ contains $\Rightarrow_t, \rightsquigarrow_t$ and $>_t$

⁷ Due to the expressive power of constraints and of the possibility of global redirections, some of the rules may depend not only on the labels or edges occurring in a term-graph, but also on the *names* of the nodes (for instance if we write a function checking whether two nodes are physically equal: $\beta: eq(\alpha, \alpha) \rightarrow \beta: true$ and $[\beta: eq(\alpha, \beta) \rightarrow \beta: false \mid \alpha \neq \beta]$).

respectively. We write $\alpha \bowtie_t^\xi \beta$ if there exists a node γ s.t. one of the following conditions holds:

$$\begin{array}{ll} \alpha \Rightarrow_t \gamma \text{ and } \beta >_t^\xi \gamma & \text{or} \quad \alpha >_t \gamma \text{ and } \beta \Rightarrow_t^\xi \gamma \\ \alpha \sqsupseteq_t \gamma \text{ and } \beta \rightsquigarrow_t^\xi \gamma & \text{or} \quad \alpha \rightsquigarrow_t \gamma \text{ and } \beta >_t^\xi \gamma \end{array}$$

Informally, these relations are intended to capture the following properties:

- $\alpha \not\bowtie_t^\xi \beta$ means that the value of the node α (i.e. the term-graphs obtained after normalization) does not depend on the node β .
- $\alpha \not\rightsquigarrow_t^\xi \beta$ means that the node β is not affected by the normalization of α .
- $\alpha \not\rightarrow_t^\xi \beta$ means that the node β is not redirected by the normalization of α .

Thus, $\alpha \bowtie_t^\xi \beta$ expresses the fact that there is a potential “conflict” between α, β (according to the considered dependency schema) which entails that these nodes should be ordered according to \succeq in the rewriting process. For instance, the condition “ $\alpha \Rightarrow_t \gamma$ and $\beta >_t^\xi \gamma$ ”, states that a rule is applicable on α that affects a node γ on which β possibly depends. Similarly, the condition “ $\alpha >_t \gamma$ and $\beta \Rightarrow_t^\xi \gamma$ ” states that a rule can be applied on a node α , on a part of the term-graph containing a node γ which may be affected by β .

It should be intuitively clear that one of the conditions in the definition of \bowtie must be satisfied if the rule currently applicable on α “interferes” with the normalization of the node β .

Each dependency schema can be associated to a (non strict) rewriting relation:

Definition 6. A ρ -matcher σ for t at a node α is said to be eligible w.r.t. an \mathcal{R} -dependency schema ξ if:

- for any node β s.t. $\alpha \bowtie_t^\xi \beta$, we have $\alpha \succeq \beta$.
- for any node $\beta \neq \alpha$ occurring in $\sigma(L)$, we have either $\beta \succ \alpha$ or $l_t(\beta) \notin \mathcal{D}$.

Definition 7. (Non Strict Rewriting) Let \mathcal{R} be a set of rewrite rules and let $\xi = (\Rightarrow^\xi, \rightsquigarrow^\xi, >^\xi)$ be an \mathcal{R} -dependency schema.

If σ is a ρ -matcher for t then we write $t \xrightarrow{(\rho, \sigma)}^\xi s$ if $s = \rho^\sigma[t]$ and σ is eligible in t w.r.t. ξ .

We write $t \xrightarrow{\rho}^\xi s$ if $t \xrightarrow{(\rho, \sigma)}^\xi s$ for some ρ -matcher σ and $t \xrightarrow{\mathcal{R}} s$ iff $t \xrightarrow{\rho}^\xi s$ for some $\rho \in \mathcal{R}$.

In particular, strict rewriting can be considered as a particular case of non strict rewriting (using a dependency schema ξ s.t. all the relations in ξ contain all the pairs of nodes in the term-graph). The next lemma shows that this rewriting relation is compatible with term-graph renaming.

Lemma 1. Let \mathcal{R} be a set of rewrite rules and let $\xi = (\Rightarrow^\xi, \rightsquigarrow^\xi, >^\xi)$ be an \mathcal{R} -dependency schema.

Let $\rho \in \mathcal{R}$. Let t, s be two term-graphs and let η be a renaming for s . Assume that $t \xrightarrow{(\rho, \sigma)}^\xi s$. Then $\eta(t) \xrightarrow{(\rho, \eta \circ \sigma)}^\xi \eta(s)$.

Several distinct rewriting relations can be defined, by choosing different \mathcal{R} -dependency schemata. The next definition states some *semantic* criteria on the dependency schema ensuring confluence of the corresponding rewriting relation.

Definition 8. An \mathcal{N} -relation Δ is said to be invariant for an \mathcal{R} -dependency schema ξ if the following holds: if σ is a ρ -matcher for t at a node α , and if $t \xrightarrow{\xi}_{(\rho,\sigma)} s$, $\beta \Delta_s \gamma$ then either $\beta \Delta_t \gamma$ or $\beta \notin \mathcal{N}(t)$ and $\alpha \Delta_t \gamma$. An \mathcal{R} -dependency schema $\xi = (\Rightarrow^\xi, \rightsquigarrow^\xi, >^\xi)$ is invariant if $\Rightarrow^\xi, \rightsquigarrow^\xi, >^\xi$ are invariant for ξ .

The next technical lemma can be considered as the key result of the present paper. It shows in some sense that two eligible matchers necessarily commute if the dependency schema is invariant. This is essential for proving confluence.

Lemma 2. Let \mathcal{R} be a set of rewrite rules and let $\xi = (\Rightarrow^\xi, \rightsquigarrow^\xi, >^\xi)$ be an invariant \mathcal{R} -dependency schema. Let ρ, ρ' be two rules in \mathcal{R} . Let t be a term-graph. Assume that there exist two eligible disjoint matchers θ, σ for t at two nodes α, β and two rules ρ, ρ' respectively.

Then there exists an eligible ρ' -matcher σ' for $\rho^\theta[t]$ and an eligible ρ -matcher θ' for $\rho'^\sigma[t]$ s.t. $\rho'^{\sigma'}[\rho^\theta[t]] = \rho^{\theta'}[\rho'^\sigma[t]]$.

Confluence is an essential property from a programming point of view, because it ensures that any object has a unique normal form. Thus no backtracking is needed during rewriting. As in [5], confluence is defined modulo renaming.

We write $t \equiv_N s$ iff there exists a renaming η for t s.t. $\eta(t) = s$ and η is the identity on N . \equiv denotes the relation \equiv_\emptyset . Informally, $t \equiv s$ states that t, s are identical (isomorphic) up to a renaming of nodes.

Definition 9. A rewrite system is said to be weak orthogonal if for any pair of rules $\rho : [L \rightarrow R \mid \phi], \pi : [L' \rightarrow R' \mid \phi']$, and for any \mathcal{N} -mapping $\sigma \in \text{sol}(\phi) \cap \text{sol}(\phi')$ compatible with L, L' s.t. $\sigma(\text{root}(L)) = \sigma(\text{root}(L'))$, we have $\sigma(R) \equiv_N \sigma(R')$, where N denotes the nodes occurring in $\sigma(L)$ and $\sigma(L')$.

Theorem 1. (Confluence of Weak Orthogonal Systems) Let \mathcal{R} be a weak orthogonal rewrite system. Let ξ be an \mathcal{R} -dependency schema. $\xrightarrow{\xi}_{\mathcal{R} \cup \equiv}$ is confluent.

As an immediate corollary, we derive the soundness of the non strict rewriting relation $\xrightarrow{\xi}_{\mathcal{R}}$ with respect to strict rewriting:

Corollary 1. Let \mathcal{R} be a weak orthogonal rewrite system. Let ξ be an \mathcal{R} -dependency schema. If $t \xrightarrow{\xi}_{\mathcal{R}} s$, $t \xrightarrow{\succ}_{\mathcal{R}} s'$ and if s' contains no defined function, then $s = s'$.

4 Computing \mathcal{R} -Dependency Schemata

The criteria defined in Section 3 are purely semantic (see Definition 8). In order to make the relation $\xrightarrow{\xi}_{\mathcal{R}}$ computable, we have to specify how to compute *effectively*

the relations $\Rightarrow_t^\xi, \rightsquigarrow_t^\xi, >_t^\xi$ in such a way that $\xi = (\Rightarrow_t^\xi, \rightsquigarrow_t^\xi, >_t^\xi)$ satisfies the conditions of Definitions 5 and 8. This is done in the present section.

We need to introduce additional definitions. Let f be a function symbol. An f -rule is a rule $[L \rightarrow R \mid \phi]$ s.t. $l_L(\text{root}(L)) = f$.

If \mathcal{R} is a rewrite system, $\geq_{\mathcal{R}}$ denotes the smallest reflexive and transitive relation s.t. $f \geq_{\mathcal{R}} g$ if an action $\beta:g$ occurs in the right hand side of an f -rule. Intuitively, $f \geq_{\mathcal{R}} g$ if g may be “called” during the evaluation of f .

If \mathcal{R} is a rewrite system, $\geq_{\mathcal{R}}^r$ denotes the smallest reflexive and transitive relation s.t. $f \geq_{\mathcal{R}}^r g$ if there exists an f -rule $[L \rightarrow R \mid \phi]$ s.t. $\text{root}(L):g \in R$. $f \geq_{\mathcal{R}}^r g$ if g is called during the evaluation of f on the same node as f .

Let \mathcal{R} be a set of rewrite rules. A rule $[L \rightarrow R \mid \phi]$ is said to *produce a side-effect* if R affects a node α occurring in L but distinct from $\text{root}(L)$ and to *perform a global redirection* if R contains a global redirection.

The set $\mathcal{SE}_{\mathcal{R}}$ (resp. $\mathcal{RD}_{\mathcal{R}}$) is the smallest set of function symbols containing any function symbol f s.t. there exists an f -rule producing a side-effect (resp. a global redirection) and s.t. $f \in \mathcal{SE}_{\mathcal{R}}$ and $g \geq_{\mathcal{R}} f$ then $g \in \mathcal{SE}_{\mathcal{R}}$ (resp. $f \in \mathcal{RD}_{\mathcal{R}}$ and $g \geq_{\mathcal{R}}^r f$ then $g \in \mathcal{RD}_{\mathcal{R}}$). We denote by $\mathcal{SE}_{\mathcal{R}}(t)$ (resp. $\mathcal{RD}_{\mathcal{R}}(t)$) the set of nodes α s.t. $l_t(\alpha) \in \mathcal{SE}_{\mathcal{R}}$ (resp. $l_t(\alpha) \in \mathcal{RD}_{\mathcal{R}}$).

Now we define our proposed dependency schema. We denote by $\xi_{\mathcal{R}}$ the function mapping each term-graph t to a tuple $(\Rightarrow_t^\xi, \rightsquigarrow_t^\xi, >_t^\xi)$. s.t.:

- $\alpha >_t^\xi \beta$ iff $\alpha \rightarrow_t \beta$ or there exists two nodes γ, δ s.t. $\delta \succeq \alpha$, $\alpha >_t^\xi \gamma$, $\delta >_t^\xi \gamma$, $\delta \in \mathcal{SE}_{\mathcal{R}}(t)$ and $\delta >_t^\xi \beta$. Informally, α depends on β if either β is reachable from α or if β may become reachable from α at some point, i.e. if there exists a node in the term-graph that can construct a “link” between α and β .
- $\alpha \Rightarrow_t^\xi \beta$ iff $\alpha \in \mathcal{SE}_{\mathcal{R}}(t)$ and $\alpha >_t^\xi \beta$. α affects β if α depends on β and if the function labeling α may perform side effects.
- $\alpha \rightsquigarrow_t^\xi \beta$ iff $\alpha = \beta$ and $\alpha \in \mathcal{RD}_{\mathcal{R}}(t)$. α redirects β if β is α and if the function labeling α performs a global redirection.

Clearly, these relations are easy to compute. The next lemma shows that they satisfy the desired properties.

Lemma 3. *Let \mathcal{R} be a rewrite system. $\xi_{\mathcal{R}}$ is an invariant \mathcal{R} -dependency schema.*

In particular, the derivation given in the Introduction can be constructed using $\xi_{\mathcal{R}}$. We give another example:

Example 3.

$$\begin{aligned} \alpha:f(\beta:0, \gamma) &\rightarrow \alpha:d, \beta:1 && (\rho_1) \\ \alpha:g(\beta, \gamma) &\rightarrow \alpha:d && (\rho_2) \\ \alpha:h(\beta, \gamma) &\rightarrow \alpha:g, \alpha \gg_1 \gamma, \alpha \gg_2 \beta && (\rho_3) \end{aligned}$$

We consider the rules above and the term-graph: $t = c(\mu_1:f(\zeta_1:0, \zeta_2:1), \mu_2:g(\zeta_2, \zeta_3:2), \mu_3:h(\zeta_3, \zeta_4:3))$. We assume that $\mu_1 \succeq \mu_2 \succeq \mu_3$. Then strict rewriting gives:

$$\begin{aligned}
t &\xrightarrow{\rho_1} c(\mu_1:d(\zeta_1:1, \zeta_2:1), \mu_2:g(\zeta_2, \zeta_3:2), \mu_3:h(\zeta_3, \zeta_4:3)) \\
&\xrightarrow{\rho_2} c(d(\zeta_1:1, \zeta_2:1), d(\zeta_2, \zeta_3:2), \mu_3:h(\zeta_3, \zeta_4:3)) \\
&\xrightarrow{\rho_3} c(d(\zeta_1:1, \zeta_2:1), d(\zeta_2, \zeta_3:2), \mu_3:g(\zeta_4:3, \zeta_3)) \\
&\xrightarrow{\rho_2} c(d(\zeta_1:1, \zeta_2:1), d(\zeta_2, \zeta_3:2), d(\zeta_4:3, \zeta_3)).
\end{aligned}$$

Now, let us apply our non strict strategy, defined by the above \mathcal{R} -dependency schema $\xi_{\mathcal{R}}$. Here $\mathcal{SE}_{\mathcal{R}} = \{f\}$ and $\mathcal{RD}_{\mathcal{R}} = \emptyset$.

Since μ_1 is labeled by a function f in $\mathcal{SE}_{\mathcal{R}}$ and since there exists a node ζ_2 s.t. $\mu_1 \rightarrow_t \zeta_2, \mu_2 \rightarrow_t \zeta_2$, we have $\mu_1 \bowtie_t^{\xi_{\mathcal{R}}} \mu_2$. Thus μ_1 must be normalized before μ_2 .

But on the other hand, there is no node λ s.t. $\mu_1 >_t^{\xi} \lambda$ and $\mu_3 >_t^{\xi} \lambda$. Indeed, the only node λ s.t. $\mu_1 >_t^{\xi} \lambda$ are ζ_1, ζ_2 and the only node λ s.t. $\mu_3 >_t^{\xi} \lambda$ are ζ_3, ζ_4 (this follows immediately from the above definition⁸). Hence μ_1, μ_3 can be normalized in arbitrary order. Thus we could reduce for instance the nodes in one of these orderings: μ_1, μ_3, μ_2 or μ_3, μ_1, μ_2 .

If \mathcal{R} is a term rewrite system (in the usual sense) then \mathcal{R} affects no node distinct from α and the constraint part of the nodes are empty. Thus we have $\alpha \not\rightarrow_t^{\xi} \beta$ and $\alpha \not\sqsupseteq_t \beta$ if $\alpha \neq \beta$. Therefore any ρ -matcher is eligible and our “flexible” rewriting relation coincides with the usual term rewriting relation.

5 Implicit Deletion of Nodes (Garbage Collection)

Node deletions can be explicitly declared by the programmer as we have seen in Section 2.3. But it is also useful to have a way of deleting (automatically) “useless” nodes, i.e. nodes that are not needed for finding the normal form of a given term-graph. Of course this can be done safely only if we are interested by the part of the term-graph that is reachable from its root (this is usually the case in practice). In this case we can delete the nodes that are not “connected” to the root (in some sense to be defined). The next definition formalizes this.

Definition 10. *Let \mathcal{R} be a rewrite system. Let ξ be an \mathcal{R} -dependency schema. Let t be a rooted term-graph. The set of useful nodes is inductively defined as follows: α is useful if either $\text{root}(t) \rightarrow_t \alpha$ or if there exists a node $\beta \in \mathcal{SE}_{\mathcal{R}}(t)$ and a useful node γ s.t. $\beta \rightarrow_t \alpha$ and $\beta \rightarrow_t \gamma$.*

If t is a term-graph, we write $t \rightarrow s$ if s is obtained from t by removing useless nodes. We denote by $\xrightarrow{\xi}_{\mathcal{R}}$ the relation $\xrightarrow{\xi}_{\mathcal{R}} \cup \rightarrow \cup \equiv$.

The next lemma states the soundness of this transformation w.r.t. rewriting.

Lemma 4. *Let \mathcal{R} be a rewrite system and let ξ be an \mathcal{R} -dependency schema. $\xrightarrow{\xi}_{\mathcal{R}}$ is confluent.*

Corollary 2. *Let t be a rooted term-graph. If $t \xrightarrow{\xi}_{\mathcal{R}} s$, then $t \xrightarrow{\xi}_{\mathcal{R}} s'$, for some $s' \rightarrow s$.*

⁸ Note that we have $\mu_2 >_t^{\xi} \zeta_1$.

6 Conclusion

Our contribution was twofold:

We introduced a conservative extension of term rewrite rules, operating on *complex data-structures*. This allows one to specify – in an abstract way – algorithms that may handle pointers. This significantly enlarges the scope of declarative languages. Since these rules are not confluent in general, their application is controlled by a *priority ordering* between the nodes.

Then we showed how to define more efficient, non deterministic strategies (including a form of garbage collection). This is done by carefully inspecting the dependencies between the nodes. Such strategies are useful because they skip some useless steps and yield shorter derivations. We showed the confluence of these strategies and their soundness w.r.t. the above priority ordering.

We would like to emphasize the fact that the concrete dependency schema presented in Section 4 is given only as an example. Other, refined relations could be defined instead (provided they fulfill the semantic conditions of Section 3).

The natural continuation of our work is to design **needed rewrite strategies** (i.e. strategies in which all rewriting steps are needed to find the normal form 10). Of course, we will need to impose additional restrictions on the considered rules (as it is done for term rewrite rules). Other, more refined, dependency schemata could also be developed (yielding more efficient rewrite strategies). For instance, one could take into account the specific features on which a given function performs a side-effect.

References

1. Ait-Kaci, H., Podelski, A.: Towards a Meaning of LIFE. *J. Log. Program* 16(3), 195–234 (1993)
2. Ariola, Z.M., Klop, J.W., Plump, D.: Bisimilarity in term graph rewriting. *Inf. Comput.* 156(1-2), 2–24 (2000)
3. Bakewell, A., Plump, D., Runciman, C.: Checking the shape safety of pointer manipulations. *RelMiCS*, pp. 48–61 (2003)
4. Barendregt, H., van Eekelen, M., Glauert, J., Kenneway, R., Plasmeijer, M.J., Sleep, M.: Term Graph Rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) *PARLE'87*. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
5. Caferra, R., Echahed, R., Peltier, N.: Rewriting term-graphs with priority. In: *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pp. 109–120. ACM Press, New York (2006)
6. Cirstea, H., Kirchner, C., Liquori, L., Wack, B.: Rewrite strategies in the rewriting calculus. *Electr. Notes Theor. Comput. Sci.* 86(4) (2003)
7. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. In: *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics (B), pp. 243–320 (1990)
8. Echahed, R., Janodet, J.-C.: Admissible graph rewriting and narrowing. In: *Proceedings of 15th International Conference and Symposium on Logic Programming*, Manchester, pp. 325–340. MIT Press, Cambridge, MA (1998)
9. Echahed, R., Peltier, N.: Narrowing data-structures with pointers. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) *ICGT 2006*. LNCS, vol. 4178, Springer, Heidelberg (2006)

10. Huet, G., Lévy, J.-J.: Computation in orthogonal rewriting systems. In: Lassez, J.-L., Plotkin, G. (eds.) *Computational Logic: Essays in Honor of Alan Robinson*, pp. 394–443. MIT Press, Cambridge, MA (1991)
11. Kennaway, J.R., Klop, J.K., Sleep, M.R., De Vries, F.J.: Transfinite Reduction in Orthogonal Term Rewriting Systems. *Information and Computation* 119(1), 18–38 (1995)
12. Plump, D.: Term graph rewriting. In: Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G.(eds.) *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 2. World Scientific (1998)
13. Plump, D.: Confluence of graph transformation revisited. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) *Processes, Terms and Cycles: Steps on the Road to Infinity*. LNCS, vol. 3838, pp. 280–308. Springer, Heidelberg (2005)
14. Schorr, H., Waite, W.M.: An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures. *Communication of the ACM* 10, 501–506 (1967)
15. Visser, E.: A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.* 40(1), 831–873 (2005)

Symbolic Model Checking of Infinite-State Systems Using Narrowing

Santiago Escobar¹ and José Meseguer²

¹ Universidad Politécnica de Valencia, Spain
sescobar@dsic.upv.es

² University of Illinois at Urbana-Champaign, USA
meseguer@cs.uiuc.edu

Abstract. Rewriting is a general and expressive way of specifying concurrent systems, where concurrent transitions are axiomatized by rewrite rules. Narrowing is a complete symbolic method for model checking reachability properties. We show that this method can be reinterpreted as a *lifting simulation* relating the original system and the symbolic system associated to the narrowing transitions. Since the narrowing graph can be infinite, this lifting simulation only gives us a semi-decision procedure for the failure of invariants. However, we propose new methods for folding the narrowing tree that can in practice result in finite systems that symbolically simulate the original system and can be used to algorithmically verify its properties. We also show how both narrowing and folding can be used to symbolically model check systems which, in addition, have state predicates, and therefore correspond to Kripke structures on which *ACTL** and *LTL* formulas can be algorithmically verified using such finite symbolic abstractions.

1 Introduction

Model checking techniques have proved enormously effective in verification of concurrent systems. However, the standard model checking algorithms only work when the set of states reachable from the given initial state is finite. Various model checking techniques for infinite-state systems exist, but they are less developed than finite-state techniques and tend to place stronger limitations on the kind of systems and/or the properties that can be model checked.

In this work we adopt the rewriting logic point of view, in which a concurrent system can always be axiomatized as a rewrite theory modulo some equational axioms, with system transitions described by rewrite rules. We then propose a new narrowing-based method for model checking such, possibly infinite-state, systems under reasonable assumptions. The key insight is that the well-known theorem on the completeness of narrowing (which for rewrite theories whose rules need not be convergent have to satisfy a topmost restriction) can be reinterpreted as a *lifting simulation* between two systems, namely, between the initial model associated to the rewrite theory (which describes our system of interest), and a “symbolic abstraction” of such a system by the narrowing relation.

The narrowing relation itself may still lead to an infinite-state system. Even then, narrowing already gives us a semi-decision procedure for finding failures of invariants. To obtain a finite-state abstraction, we then define a second simulation by *folding* the narrowing-based abstraction, using a generalization criterion to fold the possibly infinite narrowing tree into a finite graph. There is no guarantee that such a folding will always be finite. But we think that such foldings can be finite in many practical cases and give several examples of finite concurrent system abstractions of infinite systems that can be obtained in this way and can be used to verify properties of infinite systems.

Our work applies not only to the model checking of invariants, but also to the model checking of $ACTL^*$ and LTL temporal logic formulas; not just for one initial state, but for a possibly infinite, symbolically described set of initial states. We therefore also provide results about the $ACTL^*$ and LTL model checking of concurrent systems axiomatized as rewrite theories. For such temporal logic model checking we have to perform narrowing in two different dimensions: (i) in the dimension of *transitions*, as already explained above; and (ii) in the dimensions of *state predicates*, because they are not defined in general for arbitrary terms with variables, but only for suitable substitution instances. Again, our narrowing techniques, when successful in folding the system into a finite-state abstraction, allow the use of standard model checking algorithms to verify $ACTL^*$ and LTL properties of the corresponding infinite-state systems.

After some preliminaries in Section 2, we consider narrowing for model checking invariants of transition systems in Section 3, and narrowing for model checking temporal logic formulas on Kripke structures in Section 4. We conclude in Section 5. Throughout we use Lamport’s infinite-state “bakery” protocol as the source of various examples. Other examples based on a readers-writers protocol and the proofs of all technical results are included in 16.

1.1 Related Work

The idea that narrowing in its reachability sense should be used as a method for analyzing concurrent systems and should fit within a wider spectrum of analysis capabilities, was suggested in 26,13, and was fully developed in 24. The application of this idea to the verification of cryptographic protocols has been further developed by the authors in collaboration with Catherine Meadows and has been used as the basis of the Maude-NPA protocol analyzer 15. In relation to such previous work, we contribute several new ideas, including the use of lifting simulations, the folding of the narrowing graph by a generalization criterion, and the new techniques for the verification of $ACTL^*$ and LTL properties.

The methods proposed in this paper are complementary to other infinite-state model checking methods, of which narrowing is one. What narrowing has in common with various infinite-state model checking analyses is the idea of representing sets of states *symbolically*, and to perform reachability analysis to verify properties. The symbolic representations vary from approach to approach. String and multiset grammars are often used to symbolically compute

reachability sets, sometimes in conjunction with descriptions of the systems as rewrite theories [5,4], and sometimes in conjunction with learning algorithms [33]. Tree automata are also used for symbolic representation [19,30]. In general, like narrowing, some of these methods are only semi-decision procedures; but by restricting the classes of systems and/or the properties being analyzed, and by sometimes using acceleration or learning techniques, actual algorithms can be obtained for suitable subclasses: see the references above and also [6,7,14,18].

Two infinite-state model checking approaches closer in spirit to ours are: (i) the “constraint-based multiset rewriting” of Delzanno [12,11], where the infinity of a concurrent system is represented by the use of constraints (over integer or real numbers) and reachability analysis is performed by rewriting with a constraint store to which more constraints are added and checked for satisfiability or failure; and (ii) the logic-programming approach of [3], where simulations/bisimulations of labeled transition systems and symbolic representations of them using terms with variables and logic programming are studied. In spite of their similarities, the technical approaches taken in (i) and (ii) are quite different from ours. In (i), the analogue of narrowing is checking satisfiability of the constraint store; whereas in (ii) the main focus is on analyzing process calculi and on developing effective techniques using tabled logic programming to detect when a simulation or bisimulation exists.

Our work is also related to abstraction techniques, e.g., [8,22,20,21,31], which can sometimes collapse an infinite-state system into a finite-state one. In particular, it is related to, and complements, abstraction techniques for rewrite theories such as [29,23,17]. In fact, all the simulations we propose, especially the ones involving folding, can be viewed as suitable abstractions. From this point of view, our results provide new methods for automatically defining correct abstractions in a symbolic way. There is, finally, related work on computing finite representations of the search space associated by narrowing to an expression in a rewrite theory, e.g., for computing regular expressions denoting a possibly infinite set of unifiers in [2], or for partial evaluation in [1]. However, these works have a different motivation and do not consider applications to simulation/bisimulation issues, although they contain notions of correctness and completeness suitable for such applications.

2 Preliminaries

We follow the classical notation and terminology from [32] for term rewriting and from [25,27] for rewriting logic and order-sorted notions. We assume an *order-sorted signature* Σ with a finite poset of sorts (S, \leq) and a finite number of function symbols. We furthermore assume that: (i) each connected component in the poset ordering has a top sort, and for each $s \in S$ we denote by $[s]$ the top sort in the component of s ; and (ii) for each operator declaration $f : s_1 \times \dots \times s_n \rightarrow s$ in Σ , there is also a declaration $f : [s_1] \times \dots \times [s_n] \rightarrow [s]$. We assume an S -sorted family

$\mathcal{X} = \{\mathcal{X}_s\}_{s \in \mathbf{S}}$ of disjoint variable sets with each \mathcal{X}_s countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_s$ is the set of terms of sort s , and $\mathcal{T}_{\Sigma,s}$ is the set of ground terms of sort s . We write $\mathcal{T}_\Sigma(\mathcal{X})$ and \mathcal{T}_Σ for the corresponding term algebras. The set of positions of a term t is written $Pos(t)$, and the set of non-variable positions $Pos_\Sigma(t)$. The root of a term is λ . The subterm of t at position p is $t|_p$ and $t[u]_p$ is the subterm $t|_p$ in t replaced by u . A *substitution* σ is a sorted mapping from a finite subset of \mathcal{X} , written $Dom(\sigma)$, to $\mathcal{T}_\Sigma(\mathcal{X})$. The set of variables introduced by σ is $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The restriction of σ to a set of variables V is $\sigma|_V$.

A Σ -*equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_s$ for some sort $s \in \mathbf{S}$. Given Σ and a set E of Σ -equations such that $\mathcal{T}_{\Sigma,s} \neq \emptyset$ for every sort s , order-sorted equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [27]). Throughout this paper we assume that $\mathcal{T}_{\Sigma,s} \neq \emptyset$ for every sort s . The *E-subsumption* order on terms $\mathcal{T}_\Sigma(\mathcal{X})_s$, written $t \preceq_E t'$ (meaning that t' is more general than t), holds if $\exists \sigma : t =_E \sigma(t')$. The *E-renaming* equivalence on terms $\mathcal{T}_\Sigma(\mathcal{X})_s$, written $t \approx_E t'$, holds if $t \preceq_E t'$ and $t' \preceq_E t$. We extend $=_E$, \approx_E , and \preceq_E to substitutions in the expected way. An *E-unifier* for a Σ -equation $t = t'$ is a substitution σ s.t. $\sigma(t) =_E \sigma(t')$. A *complete* set of *E-unifiers* of an equation $t = t'$ is written $CSU_E(t = t')$. We say $CSU_E(t = t')$ is *finitary* if it contains a finite number of *E-unifiers*. This notion can be extended to several equations, written $CSU_E(t_1 = t'_1 \wedge \dots \wedge t_n = t'_n)$.

A *rewrite rule* is an oriented pair $l \rightarrow r$, where $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_s$ for some sort $s \in \mathbf{S}$. An (*unconditional*) *order-sorted rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ an order-sorted signature, E a set of Σ -equations, and R a set of rewrite rules. A *topmost rewrite theory* is a rewrite theory s.t. for each $l \rightarrow r \in R$, $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\text{State}}$ for a top sort **State**, $r \notin \mathcal{X}$, and no operator in Σ has **State** as an argument sort. The rewriting relation \rightarrow_R on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \xrightarrow{p}_R t'$ (or \rightarrow_R) if $p \in Pos_\Sigma(t)$, $l \rightarrow r \in R$, $t|_p = \sigma(l)$, and $t' = t[\sigma(r)]_p$ for some σ . The relation $\rightarrow_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $=_E; \rightarrow_R; =_E$. Note that $\rightarrow_{R/E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\rightarrow_{R/E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \rightarrow_{R/E} [t']_E$ iff $t \rightarrow_{R/E} t'$. When $\mathcal{R} = (\Sigma, E, R)$ is a topmost rewrite theory we can safely restrict ourselves to the rewriting relation $\rightarrow_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$, where $t \xrightarrow{A}_{R,E} t'$ (or $\rightarrow_{R,E}$) if $l \rightarrow r \in R$, $t =_E \sigma(l)$, and $t' = \sigma(r)$. Note that $\rightarrow_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\rightarrow_{R,E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \rightarrow_{R,E} [t']_E$ iff $\exists w \in \mathcal{T}_\Sigma(\mathcal{X})$ s.t. $t \rightarrow_{R,E} w$ and $w =_E t'$.

The narrowing relation \rightsquigarrow_R on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \xrightarrow{p;\sigma}_R t'$ (or $\xrightarrow{\sigma}_R, \rightsquigarrow_R$) if $p \in Pos_\Sigma(t)$, $l \rightarrow r \in R$, $\sigma \in CSU_\emptyset(t|_p = l)$, and $t' = \sigma(t[r]_p)$. Assuming that E has a finitary and complete unification algorithm, the narrowing relation $\rightsquigarrow_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ is $t \xrightarrow{p;\sigma}_{R,E} t'$ (or $\xrightarrow{\sigma}_{R,E}, \rightsquigarrow_{R,E}$) if $p \in Pos_\Sigma(t)$, $l \rightarrow r \in R$, $\sigma \in CSU_E(t|_p = l)$, and $t' = \sigma(t[r]_p)$. Note that $\rightsquigarrow_{R,E}$ on $\mathcal{T}_\Sigma(\mathcal{X})$ induces a relation $\rightsquigarrow_{R,E}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ by $[t]_E \xrightarrow{\sigma}_{R,E} [t']_E$ iff $\exists w \in \mathcal{T}_\Sigma(\mathcal{X}) : t \xrightarrow{\sigma}_{R,E} w$ and $w =_E t'$. Note that, since we will only consider topmost rewrite theories, we avoid any coherence problems, and, as pointed above for $\rightarrow_{R/E}$ and $\rightarrow_{R,E}$, the narrowing relation $\rightsquigarrow_{R,E}$ achieves the same effect as a more general narrowing relation $\rightsquigarrow_{R/E}$ (see [24]).

3 Narrowing-Based Reachability Analysis

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ specifies a transition system $\mathcal{T}_{\mathcal{R}}$ whose states are elements of the initial algebra $\mathcal{T}_{\Sigma/E}$, and whose transitions are specified by R . Before discussing the narrowing-based reachability analysis of the system $\mathcal{T}_{\mathcal{R}}$, we review some basic notions about transition systems.

Definition 1 (Transition System). A transition system is written $\mathcal{A} = (A, \rightarrow)$, where A is a set of states, and \rightarrow is a transition relation between states, i.e., $\rightarrow \subseteq A \times A$. We write $\mathcal{A} = (A, \rightarrow, I)$ when $I \subseteq A$ is a set of initial states.

Frequently, we will restrict our attention to a set of initial states in the transition system and, therefore, to the subsystem of states and transitions reachable from those initial states. However, we can obtain a useful approximation of such a reachable subsystem by using a *folding relation* in order to shrink the associated transition system, i.e., to collapse several states into a previously seen state according to some criteria.

Definition 2 (Folding Reachable Transition Subsystem). Given $\mathcal{A} = (A, \rightarrow, I)$ and a relation $G \subseteq A \times A$, the reachable subsystem from I in A with folding G is written $\text{Reach}_{\mathcal{A}}^G(I) = (\text{Reach}_{\rightarrow}^G(I), \rightarrow^G, I)$, where

$$\begin{aligned} \text{Reach}_{\rightarrow}^G(I) &= \bigcup_{n \in \mathbb{N}} \text{Frontier}_{\rightarrow}^G(I)_n, \\ \text{Frontier}_{\rightarrow}^G(I)_0 &= I, \\ \text{Frontier}_{\rightarrow}^G(I)_{n+1} &= \{y \in A \mid (\exists z \in \text{Frontier}_{\rightarrow}^G(I)_n : z \rightarrow y) \wedge \\ &\quad (\nexists k \leq n, w \in \text{Frontier}_{\rightarrow}^G(I)_k : y G w)\}, \\ \rightarrow^G &= \bigcup_{n \in \mathbb{N}} \rightarrow_{n+1}^G, \\ x \rightarrow_{n+1}^G y &\begin{cases} \text{if } x \in \text{Frontier}_{\rightarrow}^G(I)_n, y \in \text{Frontier}_{\rightarrow}^G(I)_{n+1}, x \rightarrow y; \text{ or} \\ \text{if } x \in \text{Frontier}_{\rightarrow}^G(I)_n, y \notin \text{Frontier}_{\rightarrow}^G(I)_{n+1}, \\ \quad \exists k \leq n : y \in \text{Frontier}_{\rightarrow}^G(I)_k, \exists w : (x \rightarrow w \wedge w G y) \end{cases} \end{aligned}$$

Note that, the more general the relation G , the greater the chances of $\text{Reach}_{\mathcal{A}}^G(I)$ being a finite transition system. In this paper, we consider only folding relations $G \in \{=_E, \approx_E, \preceq_E\}$ on transition systems whose state set is $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\mathfrak{s}}$ for a given sort \mathfrak{s} . We plan to study other folding relations. For $=_A = \{(a, a) \mid a \in A\}$, we write $\text{Reach}_{\mathcal{A}}(I)$ for the transition system $\text{Reach}_{\mathcal{A}}^{\preceq_A}(I)$, which is the standard notion of reachable subsystem. We are furthermore interested in comparisons between different transition systems, for which we use the notions of simulation, lifting simulation, and bisimulation.

Definition 3 (Simulation, lifting simulation, and bisimulation). Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ be two transition systems. A simulation from \mathcal{A} to \mathcal{B} , written $\mathcal{A} H \mathcal{B}$, is a relation $H \subseteq A \times B$ such that $a H b$ and $a \rightarrow_{\mathcal{A}} a'$ implies that there exists $b' \in B$ such that $a' H b'$ and $b \rightarrow_{\mathcal{B}} b'$. Given $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, I_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, I_{\mathcal{B}})$, H is a simulation from \mathcal{A} to \mathcal{B} if $(A, \rightarrow_{\mathcal{A}}) H (B, \rightarrow_{\mathcal{B}})$ and $\forall a \in I_{\mathcal{A}}, \exists b \in I_{\mathcal{B}}$ s.t. $a H b$. A simulation H from $(A, \rightarrow_{\mathcal{A}})$ to $(B, \rightarrow_{\mathcal{B}})$ (resp. from $(A, \rightarrow_{\mathcal{A}}, I_{\mathcal{A}})$ to $(B, \rightarrow_{\mathcal{B}}, I_{\mathcal{B}})$) is a bisimulation if H^{-1} is a simulation from

(B, \rightarrow_B) to (A, \rightarrow_A) (resp. from (B, \rightarrow_B, I_B) to (A, \rightarrow_A, I_A)). We call a simulation $(A, \rightarrow_A, I_A) H (B, \rightarrow_B, I_B)$ a lifting simulation if for each finite sequence $b_0 \rightarrow_B b_1 \rightarrow_B b_2 \rightarrow_B \dots \rightarrow_B b_n$ with $b_0 \in I_B$, there exists a finite sequence $a_0 \rightarrow_A a_1 \rightarrow_A a_2 \rightarrow_A \dots \rightarrow_A a_n$ with $a_0 \in I_A$ such that $a_i H b_i$ for $0 \leq i \leq n$.

Note that a lifting simulation is not necessarily a bisimulation. A lifting simulation is a simulation which ensures that false finite counterexamples do not exist. It is easy to see that simulations, lifting simulations, and bisimulations compose, that is, if $\mathcal{A} H \mathcal{B} K \mathcal{C}$ are simulations (resp. lifting simulations, resp. bisimulations), then $\mathcal{A} H; K \mathcal{C}$ is a simulation (resp. lifting simulation, resp. bisimulation). In fact, we have associated categories, with transition systems as objects and simulations (resp. lifting simulations, resp. bisimulations) as morphisms.

In rewriting logic we usually specify a concurrent system as a topmost [\[1\]](#) rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where states are E -equivalence classes of ground terms of a concrete top sort `State`, i.e., elements in $\mathcal{T}_{\Sigma/E, \text{State}}$, and transitions are rewrite rules $l \rightarrow r$ for $l, r \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\text{State}}$ that rewrite states into states. We can describe the operational behavior of the concurrent system by an associated transition system.

Definition 4 ($\mathcal{T}_{\mathcal{R}}$ -Transition System). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort `State`. We define the transition system $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E, \text{State}}, \rightarrow_{R, E})$.

Example 1. Consider a simplified version of Lamport’s bakery protocol, in which we have several processes, each denoted by a natural number, that achieve mutual exclusion between them by the usual method common in bakeries and deli shops: there is a number dispenser, and customers are served in sequential order according to the number that they hold. This system can be specified as an order-sorted topmost rewrite theory in Maude [\[2\]](#) as follows:

¹ Obviously, not all concurrent systems need to have a topmost rewrite theory specification. However, as explained in [\[24\]](#), many concurrent systems of interest, including the vast majority of distributed algorithms, admit topmost specifications. For example, concurrent object-oriented systems whose state is a multiset of objects and messages can be given a topmost specification by enclosing the system state in a top operator. Even hierarchical distributed systems of the “Russian doll” kind can likewise be so specified, provided that the boundaries defining such hierarchies are not changed by transitions.

² The Maude syntax is so close to the corresponding mathematical notation for defining rewrite theories as to be almost self-explanatory. The general point to keep in mind is that each item: a sort, a subsort, an operation, an equation, a rule, etc., is declared with an obvious keyword: `sort`, `subsort`, `op`, `eq`, `r1`, etc., with each declaration ended by a space and a period. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is defined with the signature Σ using keyword `op`, equations in E are specified using keyword `eq` or keywords `assoc`, `comm` and `id`: (for associativity, commutativity, and identity, respectively) appearing in an operator declaration, and rules in R using keyword `r1`. Another important point is the use of “mix-fix” user-definable syntax, with the argument positions specified by underbars; for example: `if_then_else_fi`. We write the sort of a variable using keyword `var` or after its name and a colon, e.g. `X:Nat`.

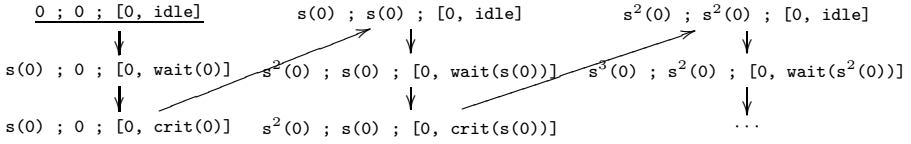


Fig. 1. Infinite transition system $\text{Reach}_{\mathcal{T}_R}(0 ; 0 ; [0, \text{idle}])$

```
fmod BAKERY-SYNTAX is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  sorts ModeIdle ModeWait ModeCrit Mode .
  subsorts ModeIdle ModeWait ModeCrit < Mode .
  sorts ProcIdle ProcWait Proc ProcIdleSet ProcWaitSet ProcSet .
  subsorts ProcIdle < ProcIdleSet .
  subsorts ProcWait < ProcWaitSet .
  subsorts ProcIdle ProcWait < Proc < ProcSet .
  subsorts ProcIdleSet < ProcWaitSet < ProcSet .
  op idle : -> ModeIdle .
  op wait : Nat -> ModeWait .
  op crit : Nat -> ModeCrit .
  op [_,_] : Nat ModeIdle -> ProcIdle .
  op [_,_] : Nat ModeWait -> ProcWait .
  op [_,_] : Nat Mode -> Proc .
  op none : -> ProcIdleSet .
  op __ : ProcIdleSet ProcIdleSet -> ProcIdleSet [assoc comm id: none] .
  op __ : ProcWaitSet ProcWaitSet -> ProcWaitSet [assoc comm id: none] .
  op __ : ProcSet ProcSet -> ProcSet [assoc comm id: none] .
  sort State .
  op _;_ : Nat Nat ProcSet -> State .
endfm
mod BAKERY is
  protecting BAKERY-SYNTAX .
  var PS : ProcSet .
  vars N M K : Nat .
  rl N ; M ; [K, idle] PS => s(N) ; M ; [K, wait(N)] PS .
  rl N ; M ; [K, wait(M)] PS => N ; M ; [K, crit(M)] PS .
  rl N ; M ; [K, crit(M)] PS => N ; s(M) ; [K, idle] PS .
endm
```

Given the initial state $t_1 = "0 ; 0 ; [0, \text{idle}]"$, where the first natural is the last distributed ticket and the second one is the value of the current ticket number accepted in critical section, the infinite transition system $\text{Reach}_{\mathcal{T}_R}(t_1)$ is depicted in Figure 1. We will graphically identify initial states by underlining them.

Narrowing calculates the most general rewriting sequences associated to a term. We can exploit this generality and use narrowing as a lifting simulation of rewriting. We write $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^\circ$ for the set of E -equivalence classes of terms of sort State excluding variables, i.e., $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^\circ = \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}} \setminus \mathcal{X}_{\text{State}}$. We can define the transition system associated to narrowing as follows.

Definition 5 (\mathcal{N}_R -Transition System). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort State . We define a transition system $\mathcal{N}_R = (\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^\circ, \rightsquigarrow_{R,E})$.

Note that we exclude variables in Definition 5, since the relation $\rightsquigarrow_{R,E}$ is not defined on them.

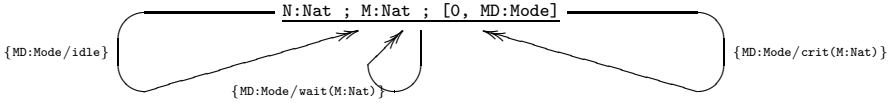


Fig. 2. Finite transition system $Reach_{\mathcal{N}_R}^{\leqslant_E}(N:Nat ; M:Nat ; [0, MD:Mode])$

Theorem [1](#) below relates the transition systems associated to narrowing and rewriting. Note that we do not have a bisimulation in general, since a term $t \in \mathcal{T}_\Sigma(\mathcal{X})$ may have narrowing steps with incomparable substitutions $\sigma_1, \dots, \sigma_k$, i.e., given $i \neq j$, $\sigma_i(t)$ may disable the rewriting step performed on $\sigma_j(t)$ and viceversa. Our results are based on the following result from [\[24\]](#).

Lemma 1 (Topmost Completeness). [\[24\]](#) *For $\mathcal{R} = (\Sigma, E, R)$ a topmost theory, let $t \in \mathcal{T}_\Sigma(\mathcal{X})$ be a term that is not a variable, and let V be a set of variables containing $Var(t)$. For some substitution ρ , let $\rho(t) \rightarrow_{R/E} t'$ using the rule $l \rightarrow r$ in R . Then there are σ, θ, t'' such that $t \overset{\sigma}{\rightsquigarrow}_{R,E} t''$ using the same rule $l \rightarrow r$, t'' is not a variable, $\rho|_V =_E (\sigma \circ \theta)|_V$, and $\theta(t'') =_E t'$.*

Given a subset $U \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_s$, we define the set of *ground instances* of U as $\llbracket U \rrbracket = \{[t]_E \in \mathcal{T}_{\Sigma/E,s} \mid \exists [t']_E \in U \text{ s.t. } t \preceq_E t'\}$. Note that U may be a finite set, whereas $\llbracket U \rrbracket$ can often be an infinite set. This gives us a symbolic way of describing possibly infinite sets of initial states in \mathcal{T}_R , which will be very useful for model checking purposes.

Theorem 1 (Lifting simulation by narrowing). *Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort $State$. Let $U \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{State}^\circ$. The relation \preceq_E defines two lifting simulations: $\mathcal{T}_R \preceq_E \mathcal{N}_R$ and $Reach_{\mathcal{T}_R}(\llbracket U \rrbracket) \preceq_E Reach_{\mathcal{N}_R}(U)$.*

Since \mathcal{N}_R is typically infinite, for a set $U \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{State}^\circ$ of initial states and a relation $G \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{State}^\circ \times \mathcal{T}_{\Sigma/E}(\mathcal{X})_{State}^\circ$, to obtain a finite abstraction we may be interested in the reachable subsystem from U in \mathcal{N}_R with folding G , i.e., in the transition system $Reach_{\mathcal{N}_R}^G(U)$.

Example 2. Consider Example [1](#) and let $t_2 = \text{“}N:Nat ; M:Nat ; [0, MD:Mode]\text{”}$. The finite transition system $Reach_{\mathcal{N}_R}^{\leqslant_E}(t_2)$ is depicted in Figure [2](#). In the case of narrowing, we will graphically tie the substitution computed by each narrowing step to the proper transition arrow. Also, when a transition step is making use of the folding relation G , i.e., when it is not a normal rewriting/narrowing step but a combination of rewriting/narrowing and folding with the relation G , we mark the arrow with a double arrowhead.

Since a transition system usually includes a set of initial states, we can extend Theorem [1](#) to a folding relation G , to obtain a more specific (and in some sense more powerful) result. For this we need the following compatibility requirement for a folding relation G .

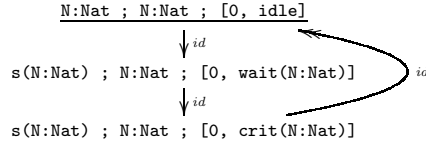


Fig. 3. Finite transition system $\text{Reach}_{\mathcal{N}_{\mathcal{R}}}^{\leq E}(\text{N:Nat ; N:Nat ; [0, idle]})$

Definition 6 ($\rightsquigarrow_{R,E}$ -equivalent relation). Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. The binary relation $G \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X}) \times \mathcal{T}_{\Sigma/E}(\mathcal{X})$ is called $\rightsquigarrow_{R,E}$ -equivalent if for $[t]_E, [t']_E, [w]_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})$ such that $t G w$ and $t \rightsquigarrow_{R,E} t'$ using rule $l \rightarrow r$, there is $[w']_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})$ such that $w \rightsquigarrow_{R,E} w'$ using rule $l \rightarrow r$ and $t' G w'$.

Lemma 2 ($\rightsquigarrow_{R,E}$ -equivalence of G). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort State . The relations $\{=_{\text{State}}, \approx_{\text{State}}, \leq_{\text{State}}\}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}$ are $\rightsquigarrow_{R,E}$ -equivalent.

Theorem 2 (Simulation by G -narrowing). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort State . Let $U \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ}$ and $G \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ} \times \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ}$ be $\rightsquigarrow_{R,E}$ -equivalent. The relation G then defines a simulation $\text{Reach}_{\mathcal{N}_{\mathcal{R}}}^G(U) G \text{Reach}_{\mathcal{N}_{\mathcal{R}}}^G(U)$.

We can obtain a bisimulation when every narrowing step of a transition system computes the identity substitution. Intuitively, every possible (ground) rewriting sequence is represented in its most general way, since narrowing does not further instantiate states in the narrowing tree. The following results rephrase Theorem 1, Lemma 2, and Theorem 2 above for bisimulations.

Theorem 3 (Bisimulation by narrowing). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort State . Let $U \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ}$. Let each transition in $\text{Reach}_{\mathcal{N}_{\mathcal{R}}}(U)$ be of the form $[t]_E \rightsquigarrow_{R,E}^{\text{id}} [t']_E$. The relation \leq_E then defines a bisimulation $\text{Reach}_{\mathcal{T}_{\mathcal{R}}}(\llbracket U \rrbracket) \leq_E \text{Reach}_{\mathcal{N}_{\mathcal{R}}}(U)$.

Lemma 3 ($\rightsquigarrow_{R,E}$ -equivalence of G^{-1}). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort State . Let $T \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}$ be such that for each $[t]_E, [t']_E \in T$, $[t]_E \rightsquigarrow_{R,E}^{\sigma} [t']_E$ implies $\sigma = \text{id}$. The relations $\{=_{\text{State}}^{-1}, \approx_{\text{State}}^{-1}, \leq_{\text{State}}^{-1}\}$ on T are $\rightsquigarrow_{R,E}$ -equivalent.

Theorem 4 (Bisimulation by G -narrowing). Let $\mathcal{R} = (\Sigma, E, R)$ be a topmost rewrite theory with a top sort State . Let $G \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ} \times \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ}$ and G^{-1} be $\rightsquigarrow_{R,E}$ -equivalent. Let $U \subseteq \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ}$. Let each transition in $\text{Reach}_{\mathcal{N}_{\mathcal{R}}}^G(U)$ be of the form $[t]_E \rightsquigarrow_{R,E}^{\text{id}} [t']_E$. The relation G then defines a bisimulation $\text{Reach}_{\mathcal{N}_{\mathcal{R}}}(U) G \text{Reach}_{\mathcal{N}_{\mathcal{R}}}^G(U)$.

Example 3. Consider Example 1 and $t_3 = \text{“N:Nat ; N:Nat ; [0, idle]”}$. The finite transition system $\text{Reach}_{\mathcal{N}_R}^{\leq E}(t_3)$ is depicted in Figure 3. Note that every transition has the *id* substitution. Therefore, by Theorems 1 and 4, we have a bisimulation between the infinite transition system $\text{Reach}_{\mathcal{T}_R}(0 ; 0 ; [0, \text{idle}])$ shown in Figure 1 and $\text{Reach}_{\mathcal{N}_R}^{\leq E}(\text{N:Nat ; N:Nat ; [0, \text{idle}]})$ in Figure 3.

Note that the narrowing-based methods we have presented allow us to answer *reachability questions* of the form $(\exists \vec{x}) t \rightarrow^* t'$. That is, given a set of initial states $\llbracket t \rrbracket$ we want to know whether from some state in $\llbracket t \rrbracket$ we can reach a state in $\llbracket t' \rrbracket$. The fact that narrowing provides a *lifting* simulation of the system \mathcal{T}_R means that it is a *complete* semi-decision procedure for answering such reachability questions: the above existential formula holds in \mathcal{T}_R if and only if from t we can reach by narrowing a term that *E*-unifies with t' . In particular, narrowing is very useful for verification of *invariants*. Let $p \in \mathcal{T}_\Sigma(\mathcal{X})_{\text{State}}$ be a pattern representing the set-theoretic complement of an invariant. Then, the reachability formula $\nexists \vec{x} : t \rightarrow^* p$ corresponds to the satisfaction of the invariant for the set of initial states $\llbracket t \rrbracket$. Therefore, narrowing provides a semi-decision procedure for the *violation* of invariants. Furthermore, the invariant holds iff p does not *E*-unify with any term in $\text{Reach}_{\mathcal{N}_R}(t)$. It also holds if p does not *E*-unify with any term in $\text{Reach}_{\mathcal{N}_R}^{\leq E}(t)$, which is a decidable question if $\text{Reach}_{\mathcal{N}_R}^{\leq E}(t)$ is finite. If p does *E*-unify with some term in $\text{Reach}_{\mathcal{N}_R}^{\leq E}(t)$, in general the invariant may or may not hold: we need to check whether this corresponds to a real narrowing sequence.

Example 4. Consider Example 1 and the following initial state with two processes $t_4 = \text{“N:Nat ; N:Nat ; [0, idle] [s(0), idle]”}$. The finite transition system $\text{Reach}_{\mathcal{N}_R}^{\leq E}(t_4)$ is depicted in Figure 4. Note that we have a bisimulation between $\text{Reach}_{\mathcal{T}_R}(\llbracket t_4 \rrbracket)$ and $\text{Reach}_{\mathcal{N}_R}^{\leq E}(t_4)$. Consider the following pattern identifying that the critical section property has been violated

$$\text{“N:Nat ; M:Nat ; [0, crit(C1:Nat)] [s(0), crit(C2:Nat)]”}.$$

We can check that the pattern does not unify with any state in the transition system of Figure 4, and thus this bad pattern is unreachable from any initial state being an instance of t_4 . This provides a verification of the *mutual exclusion* property for the infinite-state BAKERY protocol, not just from a single initial state, but from an infinite set $\llbracket t_4 \rrbracket$ of initial states.

Note, finally, that, for U a set of of initial states, even if the transition system $\text{Reach}_{\mathcal{T}_R}(\llbracket U \rrbracket)$ is finite, the transition system $\text{Reach}_{\mathcal{N}_R}^G(U)$ can be much smaller. Furthermore, the set U is typically finite, whereas the set $\llbracket U \rrbracket$ is typically infinite, making it impossible to model check an invariant from each initial state by finitary methods. In all these ways, narrowing allows algorithmic verification of invariants in many infinite-state systems, and also in finite-state systems whose size may make them unfeasible to use standard model checking techniques.

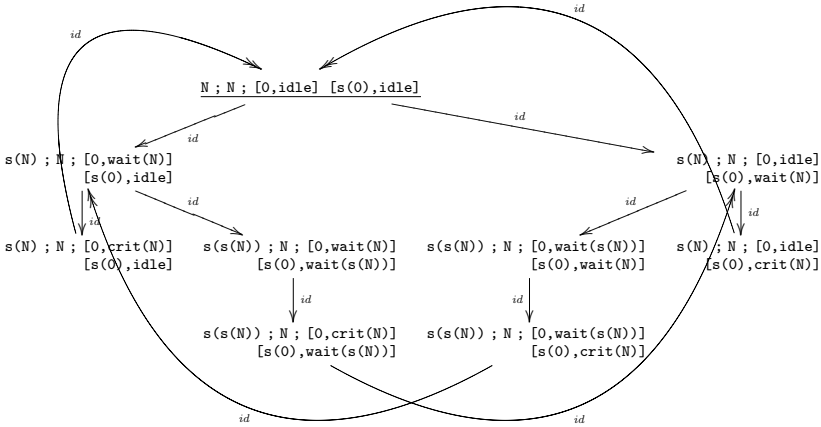


Fig. 4. Finite transition system $\text{Reach}_{\mathcal{N}^E}^{\approx} (N:\text{Nat} ; N:\text{Nat} ; [0, \text{idle}] [s(0), \text{idle}])$

4 Narrowing-Based $ACTL^*$ Model Checking

Due to space restrictions, we omit many technical details and results of this section, which can be found in [16].

Model checking [9] is the most successful verification technique for temporal logics. When we perform model checking, we use *Kripke structures* [9] to represent the state search space, which are just transition systems to which we have added a collection of atomic propositions Π on its set of states. Intuitively, in this case of model checking, for each term t with variables (denoting a symbolic state) the truth value of the atomic propositions Π may not be defined without further instantiation of t . Therefore, we cannot perform only one narrowing step in order to build the symbolic Kripke structure and must perform a narrowing step $t \rightsquigarrow_{R,E} t'$ composed with a new relation $t' \overset{\sim}{\sim}_{\Pi} \sigma(t')$ that finds an appropriate substitution σ such that the truth value of the atomic propositions in Π is entirely defined for $\sigma(t')$.

In rewriting logic we usually specify a concurrent system as a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, and the atomic propositions Π as equationally-defined predicates in an equational theory $\mathcal{E}_{\Pi} = (\Sigma_{\Pi}, E_{\Pi} \uplus E)$. As explained in Section 3, the rewrite theory \mathcal{R} contains a top sort **State**, whose data elements are E -equivalence classes in $\mathcal{T}_{\Sigma/E, \text{State}}$, and rewrite rules $l \rightarrow r \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\text{State}}$ denoting system transitions. We assume that $\Sigma_{\Pi} = \Sigma \uplus \Pi \uplus \{\mathbf{tt}, \mathbf{ff}\}$, where there is a new top sort **Bool** with no subsorts, containing only constants \mathbf{tt} and \mathbf{ff} , and each $p \in \Pi$ is an atomic proposition function symbol $p : \text{State} \rightarrow \text{Bool}$. Furthermore, we assume that each equation in E_{Π} is of the form $p(t) = \mathbf{tt}$ or $p(t) = \mathbf{ff}$, where $p \in \Pi$ and $t \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\text{State}}$, and E_{Π} is sufficiently complete and protects **Bool** (for further details see [16]).

We define a Π -Kripke structure associated to a rewrite theory \mathcal{R} and an equational theory \mathcal{E}_{Π} defining the atomic propositions Π as the triple $\mathcal{T}_{\mathcal{R}}^{\Pi} = (\mathcal{T}_{\Sigma/E, \text{State}}, (\rightarrow_{R,E})^{\bullet}, \mathcal{L}_{\Pi})$, where for each $[t]_E \in \mathcal{T}_{\Sigma/E, \text{State}}$ and $p \in \Pi$, we have

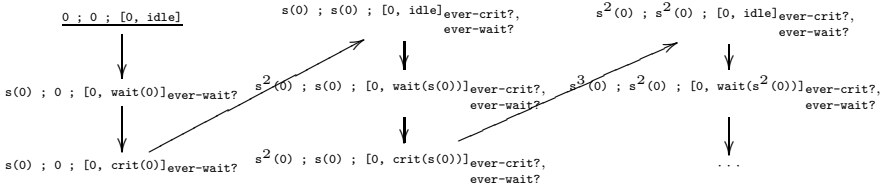


Fig. 5. Infinite Kripke structure $\mathcal{R}eac h_{\mathcal{T}_R^{\Pi}}(0 ; 0 ; [0, \text{idle}])$

$p \in \mathcal{L}_{\Pi}([t]_E) \iff p(t) =_{(E_{\Pi} \uplus E)} \text{tt}$. In what follows we will always assume that \mathcal{R} is *deadlock free*, that is, that the set of $\rightarrow_{R,E}$ -canonical forms of sort **State** is empty. As explained in [10,29], this involves no real loss of generality, since \mathcal{R} can always be transformed into a bisimilar \mathcal{R}^{df} which is deadlock free. Under this assumption the Kripke structure $\mathcal{T}_{\mathcal{R}}^{\Pi}$ then becomes the pair $\mathcal{T}_{\mathcal{R}}^{\Pi} = (\mathcal{T}_{\mathcal{R}}, \mathcal{L}_{\Pi})$. As in Section 3, given a set $U \subseteq \mathcal{T}_{\Sigma/E, \text{State}}^{\Pi}$ of initial states, we abuse the notation and define the reachable sub- Π -Kripke structure of $\mathcal{T}_{\mathcal{R}}^{\Pi}$ by $\mathcal{R}eac h_{\mathcal{T}_{\mathcal{R}}^{\Pi}}(U)$.

Example 5. Consider Example 1. We are interested in the atomic propositions $\Pi = \{\text{ever-wait?}, \text{ever-crit?}\}$ expressing that at least one process has been in its waiting (resp. critical) state.

```
fmod BAKERY-PROPS is
protecting BAKERY-SYNTAX .
sort Bool . ops tt ff : -> Bool .
ops ever-wait? ever-crit? : State -> Bool .
vars N M : Nat . vars PS : ProcSet .
eq ever-wait?(0 ; M ; PS) = ff .
eq ever-wait?(s(N) ; M ; PS) = tt .
eq ever-crit?(N ; 0 ; PS) = ff .
eq ever-crit?(N ; s(M) ; PS) = tt .
endfm
```

Given the initial state $t_1 = "0 ; 0 ; [0, \text{idle}]"$, the infinite Π -Kripke structure $\mathcal{R}eac h_{\mathcal{T}_{\mathcal{R}}^{\Pi}}(t_1)$ is depicted in Figure 5, where we would like to verify the temporal formulas " $\text{ever-wait?} \Rightarrow \Diamond \text{ever-crit?}$ " and " $\Box(\text{ever-crit?} \Rightarrow \text{ever-wait?})$ ".

As explained above, we can have symbolic states (i.e., terms $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ}$) such that the atomic propositions Π cannot be evaluated without further instantiation; check the transition system of Figure 3, where propositions `ever-wait?` and `ever-crit?` cannot be evaluated in the node "`N:Nat ; M:Nat ; [0, MD:Mode]`". We use the following relation that instantiates terms as least as possible to make propositions in Π defined.

$$t \overset{\theta}{\rightsquigarrow}_{\Pi} \theta(t) \iff \theta \in CSU_{(E_{\Pi} \uplus E)}(p_1(t) = w_1 \wedge \dots \wedge p_n(t) = w_n)$$

where for each $1 \leq i \leq n$, w_i is either `tt` or `ff`

This instantiation relation is based on whether there is a finitary and complete unification algorithm for the equational theory \mathcal{E}_{Π} , which is satisfied by the equational theories used in this paper. We can exploit the generality of narrowing and define a Kripke-structure associated to narrowing based on the following set

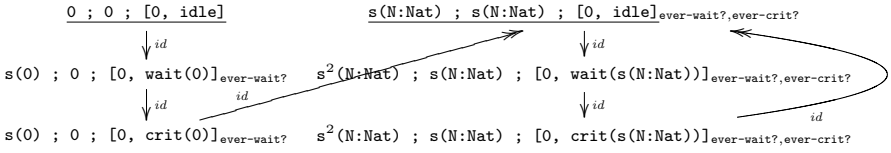


Fig. 6. Finite Kripke structure $\mathcal{R}each_{\mathcal{N}^{\Pi}}^{\mathcal{S}^E}(\{w_1, w_2\})$ with $w_1 = \text{“}0 ; 0 ; [0, \text{idle}]\text{”}$ and $w_2 = \text{“}s(N:\text{Nat}) ; s(N:\text{Nat}) ; [0, \text{idle}]\text{”}$

of terms $\mathcal{T}_{\Sigma/E}^{\Pi}(\mathcal{X})_{\text{State}}$ and the following relation $\rightsquigarrow_{R,E;\Pi}$. We define the set of terms where the truth value of Π is defined as $\mathcal{T}_{\Sigma/E}^{\Pi}(\mathcal{X})_{\text{State}}^{\circ} = \{t \in \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\text{State}}^{\circ} \mid \forall p \in \Pi : (p(t) =_{(E_{\Pi} \uplus E)} \mathbf{tt}) \vee (p(t) =_{(E_{\Pi} \uplus E)} \mathbf{ff})\}$. The narrowing relation $\rightsquigarrow_{R,E;\Pi}$ is defined as $\rightsquigarrow_{R,E}; \rightsquigarrow_{\Pi}$, i.e., $t \rightsquigarrow_{R,E;\Pi} t'$ iff $\exists w, \sigma, \sigma'$ s.t. $t \overset{\sigma}{\rightsquigarrow}_{R,E} w$, $w \overset{\sigma'}{\rightsquigarrow}_{\Pi} t'$, and $\theta = \sigma \circ \sigma'$. Note that $\rightsquigarrow_{R,E;\Pi}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ can be extended to a relation $\overset{\sigma}{\rightsquigarrow}_{R,E;\Pi}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ as $(\overset{\sigma}{\rightsquigarrow}_{R,E;\Pi}); (=E)$. We define a Kripke-structure associated to narrowing as $\mathcal{N}_{\mathcal{R}}^{\Pi} = (\mathcal{T}_{\Sigma/E}^{\Pi}(\mathcal{X})_{\text{State}}, \rightsquigarrow_{R/E;\Pi}, \mathcal{L}_{\Pi})$, where for each $[t]_E \in \mathcal{T}_{\Sigma/E}^{\Pi}(\mathcal{X})_{\text{State}}$ and $p \in \Pi$, we have $p \in \mathcal{L}_{\Pi}([t]_E) \iff p(t) =_{(E_{\Pi} \uplus E)} \mathbf{tt}$.

Results similar to Theorem 1, Lemma 2, Theorem 2, Theorem 3, Lemma 3, and Theorem 4 can be stated and proved for a deadlock-free topmost rewrite theory with a top sort **State** and a equational theory defining the atomic propositions. Such results and their proofs are included in [16].

Example 6. Consider Example 5. Consider the initial state $w = \text{“}N:\text{Nat} ; N:\text{Nat} ; [0, \text{idle}]\text{”}$, whose transition system is depicted in Figure 3. This transition system cannot be directly transformed into a Π -Kripke structure, since propositions $\{\text{ever-wait?}, \text{ever-crit?}\}$ cannot be evaluated in, for instance, state $\text{“}N:\text{Nat} ; N:\text{Nat} ; [0, \text{idle}]\text{”}$. Therefore, we must for example instantiate term w using the narrowing relation \rightsquigarrow_{Π} and obtain terms $w_1 = \text{“}0 ; 0 ; [0, \text{idle}]\text{”}$ and $w_2 = \text{“}s(N:\text{Nat}) ; s(N:\text{Nat}) ; [0, \text{idle}]\text{”}$, i.e., $w \rightsquigarrow_{\Pi} w_1$ and $w \rightsquigarrow_{\Pi} w_2$. The entire Π -Kripke structure $\mathcal{R}each_{\mathcal{N}^{\Pi}}^{\mathcal{S}^E}(\{w_1, w_2\})$ is depicted in Figure 6, where, since it is a finite-state system, we can use standard LTL model checking techniques to model check the formulas $\text{“}\text{ever-wait?} \Rightarrow \diamond \text{ever-crit?}\text{”}$ and $\text{“}\square(\text{ever-crit?} \Rightarrow \text{ever-wait?})\text{”}$, which in this case hold in $\mathcal{R}each_{\mathcal{N}^{\Pi}}^{\mathcal{S}^E}(\{w_1, w_2\})$. Therefore, the above LTL formulas also hold for the infinite-state system $\mathcal{T}_{\mathcal{R}}^{\Pi}$ of Example 5 and the infinite set $\llbracket \{w_1, w_2\} \rrbracket$ of initial states. Note that given that all substitutions in $\mathcal{R}each_{\mathcal{N}^{\Pi}}^{\mathcal{S}^E}(\{w_1, w_2\})$ are identity substitutions, we have a bisimulation and then CTL^* formulas can also be verified.

Similar arguments to those in Section 3 can be given in favor of narrowing for model checking ACTL^* (or CTL^*) properties of systems that are either infinite-state or too big for standard finite-state methods. For example, when a set $U \subseteq \mathcal{T}_{\Sigma/E}^{\Pi}(\mathcal{X})_{\text{State}}$ of initial states is provided, $\mathcal{R}each_{\mathcal{N}^{\Pi}}^G(U)$ for some G

such as \preceq_E can be finite when $\mathcal{R}each_{T^R}(\llbracket U \rrbracket)$ is infinite, or can be much smaller even in the finite-state case. And U can be finite whereas $\llbracket U \rrbracket$ may easily be infinite, making it impossible to verify properties by standard model checking algorithms.

5 Concluding Remarks

We have shown that, by specifying possibly infinite concurrent systems as rewrite theories, narrowing gives rise to a lifting simulation and provides a useful semi-decision procedure to answer reachability questions. We have also proposed a method to fold the narrowing graph that, when it yields a finite system, allows algorithmic verification of such reachability questions, including invariants. Furthermore, we have extended these techniques to the verification of *ACTL** and *LTL* formulas. Much work remains ahead, including: (i) gaining experience with many more examples such as concurrent systems, security protocols, Java program verification, etc.; (ii) implementing these techniques in *Maude*, taking advantage of its LTL model checker; (iii) investigating other folding relations that might further improve the generation of a finite narrowing search space; (iv) allowing more general state predicate definitions, for example with data parameters; (v) studying how grammar-based techniques and narrowing strategies can be used to further reduce the narrowing search space; and (vi) extending the results in this paper to more general temporal logics such as TLR [28].

Acknowledgments. We cordially thank Prasanna Thati, with whom we developed the foundations of narrowing-based reachability analysis. We also warmly thank Catherine Meadows for our joint work on narrowing-based security verification and the *Maude-NPA*; this research has provided us with a rich stimulus for investigating the new techniques presented here, which will also be very useful for security verification. S. Escobar has been partially supported by the EU (FEDER) and Spanish MEC TIN-2004-7943-C04-02 project, the Generalitat Valenciana under grant GV06/285, and the Acción Integrada Hispano-Alemana HA2006-0007. J. Meseguer's research has been supported in part by ONR Grant N00014-02-1-0715 and NSF Grant NSF CNS 05-24516.

References

1. Alpuente, M., Falaschi, M., Vidal, G.: Partial Evaluation of Functional Logic Programs. *ACM TOPLAS* 20(4), 768–844 (1998)
2. Antoy, S., Ariola, Z.M.: Narrowing the narrowing space. In: Hartel, P.H., Kuchen, H. (eds.) *PLILP 1997*. LNCS, vol. 1292, pp. 1–15. Springer, Heidelberg (1997)
3. Basu, S., Mukund, M., Ramakrishnan, C.R., Ramakrishnan, I.V., Verma, R.M.: Local and symbolic bisimulation using tabled constraint logic programming. In: Codognet, P. (ed.) *ICLP 2001*. LNCS, vol. 2237, pp. 166–180. Springer, Heidelberg (2001)

4. Bouajjani, A.: Languages, rewriting systems, and verification of infinite-state systems. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 24–39. Springer, Heidelberg (2001)
5. Bouajjani, A., Esparza, J.: Rewriting models of boolean programs. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 136–150. Springer, Heidelberg (2006)
6. Bouajjani, A., Mayr, R.: Model checking lossy vector addition systems. In: Meinel, C., Tison, S. (eds.) STACS 99. LNCS, vol. 1563, pp. 323–333. Springer, Heidelberg (1999)
7. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification over Infinite States. In: Handbook of Process Algebra, pp. 545–623. Elsevier, Amsterdam (2001)
8. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM TOPLAS 16(5), 1512–1542 (1994)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2001)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. Springer, Heidelberg (To appear 2007)
11. Delzanno, G.: Constraint multiset rewriting. Technical report, DISI - Università di Genova (2002)
12. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. STTT 3(3), 250–270 (2001)
13. Denker, G., Meseguer, J., Talcott, C.L.: Protocol specification and analysis in Maude. In: Proc. of Workshop on Formal Methods and Security Protocols (1998)
14. Emerson, A., Namjoshi, K.: On model checking for nondeterministic infinite state systems. In: LICS'98, pp. 70–80. IEEE Press, New York (1998)
15. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. Theoretical Computer Science (Elsevier) 367(1-2), 162–202 (2006)
16. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. Technical Report No. 2814, Department of Computer Science, University of Illinois at Urbana-Champaign (2007)
17. Farzan, A., Meseguer, J.: State space reduction of rewrite theories using invisible transitions. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 142–157. Springer, Heidelberg (2006)
18. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1), 63–92 (2001)
19. Genet, T., Viet Triem Tong, V.: Reachability analysis of term rewriting systems with Timbuk. In: ICLP'01. LNCS, vol. 2250, pp. 695–706. Springer, Heidelberg (2001)
20. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
21. Kesten, Y., Pnueli, A.: Control and data abstraction: The cornerstones of practical formal verification. STTT 4(2), 328–342 (2000)
22. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6, 1–36 (1995)
23. Martí-Oliet, N., Meseguer, J., Palomino, M.: Theoroidal maps as algebraic simulations. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 126–143. Springer, Heidelberg (2005)

24. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *High-Order Symbolic Computation* (To appear 2007)
25. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
26. Meseguer, J.: Multiparadigm logic programming. In: Kirchner, H., Levi, G. (eds.) *ALP'92*. LNCS, vol. 632, pp. 158–200. Springer, Heidelberg (1992)
27. 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)
28. Meseguer, J.: *The Temporal Logic of Rewriting*. Technical Report No. 2815, Department of Computer Science, University of Illinois at Urbana-Champaign (2007)
29. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. In: Baader, F. (ed.) *Automated Deduction – CADE-19*. LNCS (LNAI), vol. 2741, pp. 2–16. Springer, Heidelberg (2003)
30. Ohsaki, H., Seki, H., Takai, T.: Recognizing boolean closed A-tree languages with membership conditional mechanism. In: Nieuwenhuis, R. (ed.) *RTA 2003*. LNCS, vol. 2706, pp. 483–498. Springer, Heidelberg (2003)
31. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D.A. (eds.) *CAV 1999*. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999)
32. TeReSe, (ed.): *Term Rewriting Systems*. Cambridge University Press, Cambridge (2003)
33. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 45–60. Springer, Heidelberg (2005)

Delayed Substitutions

José Espírito Santo*

Departamento de Matemática
Universidade do Minho
Portugal
jes@math.uminho.pt

Abstract. This paper investigates an approach to substitution alternative to the implicit treatment of the λ -calculus and the explicit treatment of explicit substitution calculi. In this approach, substitutions are delayed (but not executed) explicitly. We implement this idea with two calculi, one where substitution is a primitive construction of the calculus, the other where substitutions is represented by a β -redex. For both calculi, confluence and (preservation of) strong normalisation are proved (the latter fails for a related system due to Revesz, as we show). Applications of delayed substitutions are of theoretical nature. The strong normalisation result implies strong normalisation for other calculi, like the computational lambda-calculus, lambda-calculi with generalised applications, or calculi of cut-elimination for sequent calculus. We give an investigation of the computational interpretation of cut-elimination in terms of generation, execution, and delaying of substitutions, paying particular attention to how generalised applications improve such interpretation.

1 Introduction

Explicit substitution calculi were introduced as an improvement of the λ -calculus, capable of modelling the actual implementation of functional languages and symbolic systems [1]. However, other applications of theoretical nature were soon recognized, particularly in proof theory, where λ -calculus also fails to give a computational interpretation to sequent calculus and cut-elimination [4, 5, 14].

The basic idea in explicit substitution calculi is the separation between the generation and the execution of substitution. But this idea is operative only if this execution can be delayed. Of course, the mentioned separation gives the opportunity to do something between the generation and the execution of a substitution. But there are situations, for instance in a syntax like that of the λx -calculus [12], where explicit rules for the delaying of substitution are required.

This paper investigates explicit rules for delaying substitution in a syntax similar to λx . However, a first and immediate observation is that explicit delaying cannot be combined with explicit execution without breaking termination. The situation is even worse if we try to implement substitutions as β -redexes (Revesz's idea [11]). So, the system we study, named λs , separates generation

* The author is supported by FCT via Centro de Matematica, Universidade do Minho.

and execution of substitution, but employs implicit execution. In addition, it has permutation rules for achieving the delaying of substitution.

The calculus $\lambda\mathbf{s}$ enjoys good properties, like confluence and (preservation of) strong normalisation. The circumstance of employing implicit substitution disallows direct applications of $\lambda\mathbf{s}$ to the implementation of computational systems. However, $\lambda\mathbf{s}$ has several theoretical uses. Strong normalisation of $\lambda\mathbf{s}$ implies the same property for several calculi, like the computational lambda-calculus [9] and lambda-calculi with generalised applications [6]. Certainly, future work should exploit the use of the calculus for reasoning about programs. In this paper we emphasize applications to proof theory.

We define a sequent calculus LJ with a simple cut-elimination procedure consisting of 3 reduction rules. Then, we show that $\lambda\mathbf{s}$ gives a computational interpretation to the 3 cut-elimination rules of LJ , precisely as rules for the generation, delaying, and execution of substitution. Strong normalisation of $\lambda\mathbf{s}$ is lifted to LJ . We pay particular attention to how generalised applications [6][15], when combined with delayed substitutions, improve the mentioned interpretation of LJ .

Notations: Types (=formulas) are ranged over by A, B, C and generated from type variables using the “arrow type” (=implication), written $A \supset B$. Contexts Γ are sets of declarations $x : A$ where each variable is declared at most once. Barendregt’s variable convention is adopted. In particular, we take renaming of bound variables for granted. Meta substitution is denoted by $[-/x]$. By a *value* we mean a variable or λ -abstraction in the calculus at hand.

2 Delayed Substitutions

Motivation: Recall the syntax of the $\lambda\mathbf{x}$ -calculus:

$$M, N, P ::= x \mid \lambda x.M \mid MN \mid \langle N/x \rangle M$$

The variable x is bound in M in $\lambda x.M$ and $\langle N/x \rangle M$. The scope of $\lambda x.$ and $\langle N/x \rangle.$ extends to the right as much as possible. There is a reduction rule

$$(\beta) (\lambda x.M)N \rightarrow \langle N/x \rangle M$$

that generates substitutions and four rules

$$\begin{array}{ll} (\mathbf{x}_1) \langle N/x \rangle x \rightarrow N & (\mathbf{x}_3) \langle N/x \rangle MP \rightarrow (\langle N/x \rangle M)\langle N/x \rangle P \\ (\mathbf{x}_2) \langle N/x \rangle y \rightarrow y, y \neq x & (\mathbf{x}_4) \langle N/x \rangle \lambda y.M \rightarrow \lambda y.\langle N/x \rangle M \end{array}$$

for the explicit execution of substitution. By variable convention, $x \neq y$ and $y \notin N$ in rule (\mathbf{x}_4) . Let $\mathbf{x} = \cup_{i=1}^4 \mathbf{x}_i$.

Suppose we want to reduce $Q_0 = (\lambda x.M)NN'$, where $M = \lambda y.P$. After a β -step, we obtain $Q_1 = (\langle N/x \rangle M)N'$. Substitution $\langle N/x \rangle M$ was generated but not immediately executed. This allows the *delaying* of its execution, if we

decide to do something else, e.g. reducing N , M , N' , or another term in the program surrounding Q_1 . However, we may very well be interested in delaying the execution of $\langle N/x \rangle M$ in another way, namely by applying immediately M to N' . In λx this may be achieved in some sense, if a step of the execution of $\langle N/x \rangle M$ is performed, yielding $Q_2 = (\lambda y.P')N'$, where $P' = \langle N/x \rangle P$.

This lack of separation between substitution execution and delaying is unsatisfactory. The delaying of $\langle N/x \rangle M$ in Q_1 can be achieved if we adopt a permutation rule that yields $\langle N/x \rangle MN'$, that is $Q'_2 = \langle N/x \rangle (\lambda y.P)N'$. However, we cannot add this permutation to the set of x -rules without breaking termination. Suppose we want to reduce $\langle N/x \rangle M_1 M_2$, where M_2 is a pure term (i.e. a term without substitutions) and $x \notin M_2$. Then a cycle is easily generated:

$$\begin{aligned} \langle N/x \rangle M_1 M_2 &\rightarrow_x (\langle N/x \rangle M_1) \langle N/x \rangle M_2 \text{ (by } \mathbf{x}_3) \\ &\rightarrow_x^* (\langle N/x \rangle M_1) [N/x] M_2 \text{ (because } M_2 \text{ is pure)} \\ &= (\langle N/x \rangle M_1) M_2 \text{ (because } x \notin M_2) \\ &\rightarrow \langle N/x \rangle M_1 M_2 \text{ (by permutation)} \end{aligned}$$

(Here $[N/x]M_2$ denotes meta-substitution.) This is why the calculus of delayed substitutions we introduce next does not have x -rules for explicit, stepwise execution of substitution, but instead a single σ -rule for its implicit execution.

The λs -calculus: The terms of λs are given by:

$$M, N, P, Q ::= x \mid \lambda x.M \mid MN \mid \langle N/x \rangle M$$

This set of terms is equipped with the following reduction rules:

$$\begin{array}{ll} (\beta) \ (\lambda x.M)N \rightarrow \langle N/x \rangle M & (\pi_1) \ (\langle N/x \rangle M)P \rightarrow \langle N/x \rangle MP \\ (\sigma) \ \langle N/x \rangle M \rightarrow [N/x]M & (\pi_2) \ \langle \langle N/x \rangle P/y \rangle M \rightarrow \langle N/x \rangle \langle P/y \rangle M \end{array}$$

where meta-substitution $[N/x]M$ is defined as expected. In particular

$$[N/x] \langle P/y \rangle M = \langle [N/x]P/y \rangle [N/x]M \ . \quad (1)$$

By variable convention, $x \neq y$ in π_2 and \mathbf{II} . For the same reason, $y \notin N$ in \mathbf{II} .

Let $\pi = \pi_1 \cup \pi_2$. The choice of permutations π_1 and π_2 is pragmatic. These are the permutations appropriate for the applications of the calculus to be shown in this paper. It is natural that, if the applications of the calculus are different, other rules for pulling out substitutions in other contexts are useful and needed.

By a *typable* term $M \in \lambda$ or $M \in \lambda s$ we mean a term that has a simple type A , given a context Γ assigning types to the free variable of M . This relation is written $\Gamma \vdash M : A$ and generated by the set of usual rules for assigning simple types to variables, abstraction and application (which we omit), plus the typing rule of substitution:

$$\frac{\Gamma \vdash N : A \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \langle N/x \rangle M : B}$$

Natural relationship with the λ -calculus: The study of the natural interpretation of λs in λ yields easily a proof of confluence for λs . First, λs simulates β -reduction.

Proposition 1. *If $M \rightarrow_{\beta} N$ in λ then $M \rightarrow_{\beta} P \rightarrow_{\sigma} N$ in $\lambda\mathbf{s}$, for some P .*

Conversely, let $(_)^{\natural} : \lambda\mathbf{s} \rightarrow \lambda$ be defined as follows: $x^{\natural} = x$, $(MN)^{\natural} = M^{\natural}N^{\natural}$, $(\lambda x.M)^{\natural} = \lambda x.M^{\natural}$, and $(\langle N/x \rangle M)^{\natural} = [N^{\natural}/x]M^{\natural}$.

Proposition 2. (1) *If $M \rightarrow_{\beta} N$ in $\lambda\mathbf{s}$ then $M^{\natural} \rightarrow_{\beta}^* N^{\natural}$ in λ .* (2) *If $M \rightarrow_{\pi\sigma}$ in $\lambda\mathbf{s}$ then $M^{\natural} = N^{\natural}$.* (3) *In $\lambda\mathbf{s}$, $M \rightarrow_{\sigma}^* M^{\natural}$.*

Proposition 3 (Confluence). *Let $R \in \{\pi_1, \pi_2, \pi\}$. Then $\rightarrow_{\beta\sigma R}$ is confluent in $\lambda\mathbf{s}$.*

Proof: By confluence of β -reduction in λ and Propositions [1](#) and [2](#). ■

We will prove strong normalisation of $\lambda\mathbf{s}$ as a corollary to strong normalisation of another calculus of delayed substitutions. The terms of the latter are the ordinary λ -terms, where β -redexes are regarded as substitutions. The first author to develop this idea was G. Revesz, see for instance [\[11\]](#).

Revesz's system: G. Revesz proposed to replace in the λ -calculus the ordinary β -rule and the related calls to meta-substitution by a set of local transformation rules. These local rules correspond to the explicit, stepwise execution of substitution.

$$\begin{array}{ll} (\beta_1) (\lambda x.x)N \rightarrow N & (\beta_3) (\lambda x.\lambda y.M)N \rightarrow \lambda y.(\lambda x.M)N \\ (\beta_2) (\lambda x.y)N \rightarrow y, y \neq x & (\beta_4) (\lambda x.MP)N \rightarrow (\lambda x.M)N((\lambda x.P)N) \end{array}$$

Let $\mathbf{R} = \cup_{i=1}^4 \beta_i$. By variable convention, $x \neq y$ and $y \notin N$ in β_3 . A basic property of Revesz's system is that

$$(\lambda x.M)N \rightarrow_{\mathbf{R}}^* [N/x]M . \quad (2)$$

Now, we have seen that in a syntax with a primitive substitution construction, we cannot combine explicit substitution execution and delaying without breaking termination. When substitution is represented by β -redexes, the situation is even worse, as substitution execution alone breaks termination.

Theorem 1. *There is a typed λ -term Q such that Q is not $\mathbf{R} - SN$.*

Proof: Let $Q = (\lambda x.(\lambda y.M)N)P$. We underline the successive redexes.

$$\begin{aligned} Q &= \underline{(\lambda x.(\lambda y.M)N)P} \\ &\rightarrow_{\beta_4} \underline{(\lambda x.\lambda y.M)P}((\lambda x.N)P) = Q' \\ &\rightarrow_{\beta_3} \underline{(\lambda y.(\lambda x.M)P)}((\lambda x.N)P) \\ &\rightarrow_{\beta_4} \underline{(\lambda y.\lambda x.M)}((\lambda x.N)P)\underline{((\lambda y.P))((\lambda x.N)P)} \\ &\rightarrow_{\mathbf{R}}^* \underline{(\lambda y.\lambda x.M)}((\lambda x.N)P)\underline{[(\lambda x.N)P/y]P} \quad (\text{by } \text{[\(2\)](#)}) \\ &= \underline{(\lambda y.\lambda x.M)}((\lambda x.N)P)P \quad (\text{as } y \notin P) \\ &\rightarrow_{\beta_3} \underline{(\lambda x.(\lambda y.M))}((\lambda x.N)P)\underline{P} \\ &\rightarrow_{\beta_4} \underline{(\lambda x.\lambda y.M)P}(\underline{((\lambda x.((\lambda x.N)P))P)}) \\ &\rightarrow_{\mathbf{R}}^* \underline{(\lambda x.\lambda y.M)P}[\underline{P/x}]((\lambda x.N)P) \quad (\text{by } \text{[\(2\)](#)}) \\ &= \underline{(\lambda x.\lambda y.M)P}((\lambda x.N)P) \quad (\text{as } x \notin ((\lambda x.N)P)) \\ &= Q' \end{aligned}$$
■

So, one has to give up the idea of explicitly executing substitution within the syntax of the λ -calculus. But we can do explicit delaying.

Delaying of substitution in the λ -calculus: In the λ -calculus, define $\pi = \pi_1 \cup \pi_2$, where:

$$\begin{aligned} (\pi_1) \quad & (\lambda x.M)NP \rightarrow (\lambda x.MP)N \\ (\pi_2) \quad & M((\lambda x.P)N) \rightarrow (\lambda x.MP)N \end{aligned}$$

In both redexes, M “wants” to be applied to P , but something forbids this application, namely the fact that one of M or P is inside a β -redex (or “substitution”). The rules rearrange the term so that the “substitution” is delayed. We denote the calculus consisting of β and π as $\lambda[\beta\pi]$. Notice that π_1 is one of Regnier’s σ -rules [10].

Proposition 4. *In λ , $\rightarrow_{\beta\pi}$ is confluent, but \rightarrow_{π} is not.*

Proof: Notice that $(\lambda x.M)NP =_{\beta} (\lambda x.MP)N$ and $M((\lambda y.P)N) =_{\beta} (\lambda y.MP)N$. So, confluence of $\rightarrow_{\beta\pi}$ follows from confluence of \rightarrow_{β} . On the other hand, $(\lambda x.M)N((\lambda y.P)Q)$ π -reduces to both $(\lambda x.(\lambda y.MP)Q)N$ and $(\lambda y.(\lambda x.MP)N)Q$, which can easily be two π -nfs. ■

Define $|M|$, the *size* of λ -term M , as follows: $|x| = 1$; $|\lambda x.M| = 1 + |M|$; $|MN| = 1 + |M| + |N|$.

Proposition 5. *In λ , \rightarrow_{π} is terminating.*

Proof: The termination of \rightarrow_{π_1} is in [10]. As to the remaining cases, define $w(M)$, the *weight* of a λ -term M , as follows: $w(x) = 0$; $w(\lambda x.M) = w(M)$; $w(MN) = |N| + w(M) + w(N)$. It holds that, if $M \rightarrow_{\pi_1} N$, then $w(M) = w(N)$; and that, if $M \rightarrow_{\pi_2} N$, then $w(M) > w(N)$. The proposition now follows. ■

Let M be a λ -term such that M is β -SN. Define $\|M\|_{\beta}$ to be the maximal length of β -reduction sequences starting from M .

Proposition 6. *Let $M \rightarrow_{\pi} N$. If M is β -SN, then so is N and $\|M\|_{\beta} \geq \|N\|_{\beta}$.*

Proof: For π_1 , [10] proves $\|M\|_{\beta} = \|N\|_{\beta}$. For π_2 , the argument is a slight generalisation of an argument in [7], and uses the fact that, for M, N λ -terms,

$$x \in FV(M) \Rightarrow \|(\lambda x.M)N\|_{\beta} \leq \|[N/x]M\|_{\beta} + 1 \quad (3)$$

$$x \notin FV(M) \Rightarrow \|(\lambda x.M)N\|_{\beta} \leq \|M\|_{\beta} + \|N\|_{\beta} + 1 \quad (4)$$

This is called the “fundamental lemma of perpetuality”¹. Let $Q_0 = M((\lambda x.P)N)$ and $Q_1 = (\lambda x.MP)N$. If $x \in P$, then $\|Q\|_{\beta} \geq 1 + \|M([N/x]P)\|_{\beta} \geq \|Q_1\|_{\beta}$.

¹ One immediate consequence of this fact is that if (i) $(\lambda x.M)N \notin \beta$ -SN and (ii) $N \in \beta$ -SN when $x \notin FV(M)$, then $[N/x]M \notin \beta$ -SN. It is this latter fact that is called “fundamental lemma of perpetuality” in [13].

The first inequality is by $Q_0 \rightarrow_\beta M([N/x]P)$ and the second by (3). If $x \notin P$, then $\|Q\|_\beta \geq 1 + \|N\|_\beta + \|MP\|_\beta \geq \|Q_1\|_\beta$. The first inequality is by $Q_0 \xrightarrow[\beta]{k} M((\lambda x.P)N') \rightarrow_\beta MP$ (where $k = \|N\|_\beta$) and the second by (4). ■

Theorem 2 (SN and PSN). *If $M \in \lambda$ is β -SN (in particular, if M is typable) then M is $\beta\pi$ -SN.*

Proof: From Propositions (5) and (6). ■

Sharper relationship with the λ -calculus: Let $(-)^{\sharp} : \lambda\mathfrak{s} \rightarrow \lambda[\beta\pi]$ be defined as follows: $x^{\sharp} = x$, $MN^{\sharp} = M^{\sharp}N^{\sharp}$, $(\lambda x.M)^{\sharp} = \lambda x.M^{\sharp}$, and $(\langle N/x \rangle M)^{\sharp} = (\lambda x.M^{\sharp})N^{\sharp}$. Hence, mapping $(-)^{\sharp}$ “raises” substitutions to β -redexes.

Proposition 7. (1) *If $M \rightarrow_\beta N$ in $\lambda\mathfrak{s}$ then $M^{\sharp} = N^{\sharp}$.* (2) *If $M \rightarrow_{\pi_i} N$ in $\lambda\mathfrak{s}$ then $M^{\sharp} \rightarrow_{\pi_i} N^{\sharp}$ in $\lambda[\beta\pi]$ ($i = 1, 2$).* (3) *If $M \rightarrow_\sigma N$ in $\lambda\mathfrak{s}$ then $M^{\sharp} \rightarrow_\beta N^{\sharp}$ in $\lambda[\beta\pi]$.*

Proposition 8. *In $\lambda\mathfrak{s}$, $\rightarrow_{\beta\pi}$ is terminating.*

Proof: From termination of \rightarrow_β in $\lambda\mathfrak{s}$, parts 1. and 2. of Proposition (7) and Proposition (5). ■

Proposition 9. *Let $M \in \lambda\mathfrak{s}$ and suppose M^{\sharp} is β -SN. Then M is $\beta\pi\sigma$ -SN.*

Proof: From Propositions (7), (8) and (6). ■

Theorem 3 (SN and PSN).

1. *If $M \in \lambda\mathfrak{s}$ is typable then M is $\beta\pi\sigma$ -SN.*
2. *If $M \in \lambda$ is β -SN then, in $\lambda\mathfrak{s}$, M is $\beta\pi\sigma$ -SN.*

Proof: 1. Suppose $M \in \lambda\mathfrak{s}$ is typable. Then M^{\sharp} is typable, because $(-)^{\sharp}$ preserves typability. Hence M^{\sharp} is β -SN, by strong normalisation of the simply typed λ -calculus. By Proposition (9), M is $\beta\pi\sigma$ -SN.

2. If $M \in \lambda$, then $M^{\sharp} = M$. Now apply Proposition (9). ■

3 Related Calculi

Substitution is a natural interpretation for let-expressions and generalised applications. These interpretations allow strong normalisation of $\lambda\mathfrak{s}$ to be transferred to the computational λ -calculus $\lambda_{\mathcal{C}}$ (8) and to the λJ -calculus (6).

Computational λ -calculus: Its terms are given by:

$$M, N, P ::= x \mid \lambda x.M \mid MN \mid \text{let } x = N \text{ in } M \ .$$

It was proved in (9) that strong normalisation for Moggi’s original reduction rules is a consequence of strong normalisation for the following restricted set of rules (where V stands for a value):

$$\begin{array}{ll}
 (C_1) & (\lambda x.M)V \rightarrow [V/x]M \\
 (C_2) & \text{let } x = V \text{ in } M \rightarrow [V/x]M \\
 (C_3) & \text{let } x = M \text{ in } x \rightarrow M \\
 (C_4) & \text{let } y = \text{let } x = P \text{ in } M \text{ in } N \rightarrow \text{let } x = P \text{ in let } y = M \text{ in } N \\
 (\eta_v) & \lambda x.Vx \rightarrow V, x \notin V
 \end{array}$$

Let $C = \cup_{i=1}^4 C_i$. Checking [9] again one sees that strong normalisation of \rightarrow_C (η_v dropped) is sufficient for strong normalisation of λ_C with η_v omitted.

Now, instead of mapping the restricted calculus to the linear λ -calculus (as in [9]), we simply interpret into $\lambda\mathbf{s}$, reading $\text{let } x = N \text{ in } M$ as $\langle N/x \rangle M$. With this interpretation, C_2 and C_3 are particular cases of σ , C_4 is π_2 , and C_1 is β followed by σ . Thus strong normalisation of $\lambda\mathbf{s}$ implies strong normalisation of \rightarrow_C , and, therefore, of λ_C with η_v omitted.

λ -calculus with generalised application: The system ΛJ of [6] is renamed here as $\lambda\mathbf{g}$. Terms of $\lambda\mathbf{g}$ are given by

$$M, N, P ::= x \mid \lambda x.M \mid M(N, x.P) .$$

The typing rule for generalized application is

$$\frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A \quad \Gamma, x : B \vdash P : C}{\Gamma \vdash M(N, x.P) : C} \text{ gElim}$$

The $\lambda\mathbf{g}$ -calculus has two reduction rules:

$$\begin{array}{l}
 (\beta) \quad (\lambda x.M)(N, y.P) \rightarrow [[N/x]M/y]P \\
 (\pi) \quad M(N, x.P)(N', y.P') \rightarrow M(N, x.P(N', y.P')) .
 \end{array}$$

The natural mapping $(-)^* : \lambda\mathbf{g} \rightarrow \lambda\mathbf{s}$ is given by $x^* = x$, $(\lambda x.M)^* = \lambda x.M^*$, and $(M(N, x.P))^* = \langle M^*N^*/x \rangle P^*$. This mapping gives a strict simulation (one step mapped to one or more steps). Here is the simulation of π .

$$\begin{aligned}
 (M_0(N_1, x.P_1)(N_2, y.P_2))^* &= \langle \langle \langle M_0^*N_1^*/x \rangle P_1^* \rangle N_2^*/y \rangle P_2^* \\
 &\rightarrow_{\pi_1} \langle \langle M_0^*N_1^*/x \rangle P_1^* N_2^*/y \rangle P_2^* \\
 &\rightarrow_{\pi_2} \langle M_0^*N_1^*/x \rangle \langle P_1^*N_2^*/y \rangle P_2^* \\
 &= (M_0(N_1, x.P_1(N_2, y.P_2)))^* .
 \end{aligned}$$

So, strong normalisation of $\lambda\mathbf{s}$ implies strong normalisation of $\lambda\mathbf{g}$.

4 Applications to Proof Theory

Summary of the section: The λ -calculus is the computational interpretation of natural deduction (in the setting of intuitionistic implicational logic). A λ -term is assigned to each natural deduction by the Curry-Howard correspondence, so that the interpretation of normalisation is β -reduction. There is a traditional assignment $(-)^{\flat}$ of λ -terms to sequent calculus derivations, but this assignment

fails to give a computational interpretation to the process of cut-elimination. We try an obvious assignment $(_)\nabla$ of $\lambda\mathbf{s}$ -terms, but only an optimization $(_)\blacktriangledown$ of the latter gives a computational interpretation in terms of generation, execution and delaying of substitution. As a by product, we lift strong normalisation of $\lambda\mathbf{s}$ to sequent calculus. The need for the mentioned optimization is caused by a problem of “imperfect substitution” in sequent calculus, which does not show up when translating sequent calculus with $(_)\flat$. A tool that we use in analyzing, and in some sense overcoming, this problem is the calculus $\lambda\mathbf{gs}$, a calculus with generalised application and a primitive substitution construction. Figure 1 shows the systems and mappings studied in this section.

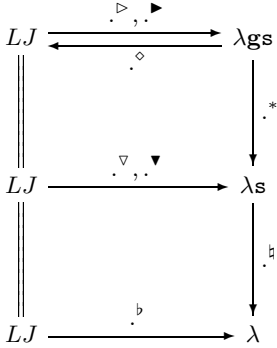


Fig. 1. Sequent calculus and delayed substitution

The calculus LJ: Sequent calculus derivations are represented by:

$$L ::= \text{Axiom}(x) \mid \text{Left}(y, L, (x)L) \mid \text{Right}((x)L) \mid \text{Cut}(L, (y)L)$$

Typing rules are as follows:

$$\frac{}{\Gamma, x : A \vdash \text{Axiom}(x) : A} \text{Axiom} \quad \frac{\Gamma \vdash L_1 : A \quad \Gamma, x : A \vdash L_2 : C}{\Gamma \vdash \text{Cut}(L_1, (x)L_2) : C} \text{Cut}$$

$$\frac{\Gamma \vdash L_1 : A \quad \Gamma, x : B \vdash L_2 : C}{\Gamma, y : A \supset B \vdash \text{Left}(y, L_1, (x)L_2) : C} \text{Left} \quad \frac{\Gamma, x : A \vdash L : B}{\Gamma \vdash \text{Right}((x)L) : A \supset B} \text{Right}$$

where $x \notin \Gamma$ in *Left*, *Right* and *Cut*.

Cut elimination in LJ follows the *t*-protocol of [2]. If a cut is right-permutable, perform its complete, upward, right permutation. This is the first structural step (*S1*). If a cut is not right-permutable, but is left-permutable, perform its complete, upward, left permutation. This is the second structural step (*S2*). If a cut is neither right-permutable, nor left-permutable, then it is a logical cut (both cut formulas main in the premisses). In that case, apply the logical step of cut-elimination (*Log*), generating cuts with simpler cut-formula.

Let the predicate $mll(x, L)$ (read “ x is main in a linear left introduction L ”) be defined by: $mll(x, L)$ iff there are L_1, y, L_2 such that $L = \text{Left}(x, L_1, (y)L_2)$ and $x \notin L_1, L_2$. The reduction rules of LJ are as follows (where Axiom, Left, Right and Cut are abbreviated by A, L, R and C, respectively):

$$\begin{aligned}
 (S1) & \quad C(L_1, (x)L_2) \rightarrow S1(L_1, x, L_2) \\
 (S20) & \quad C(A(y), (x)L(x, L'_1, (y')L'_2)) \rightarrow L(y, L'_1, (y')L'_2) \\
 (S21) & \quad C(L(z, L_1, y, L_2), (x)L(x, L'_1, (y')L'_2)) \rightarrow L(z, L_1, y, S2(L_2, x, L'_1, y', L'_2)) \\
 (S22) & \quad C(C(L_1, (y)L_2), (x)L(x, L'_1, (y')L'_2)) \rightarrow C(L_1, y, S2(L_2, x, L'_1, y', L'_2)) \\
 (Log) & \quad C(R((y)L_1), (x)L(x, L'_1, (y')L'_2)) \rightarrow C(C(L'_1, (y)L_1), (y')L'_2)
 \end{aligned}$$

Provisos: in $S1$, not $mll(x, L_2)$; in the remaining rules, $x \notin L'_1, L'_2$. We let $S2 = \cup_{i=0}^2 S2i$. The meta-operations $S1$ and $S2$ are given by:

$$\begin{aligned}
 S1(L, x, \text{Axiom}(x)) &= L \\
 S1(L, x, \text{Axiom}(y)) &= \text{Axiom}(y), y \neq x \\
 S1(L, x, \text{Left}(x, L', (z)L'')) &= \text{Cut}(L, (x)\text{Left}(x, S1(L, x, L'), (z)S1(L, x, L'')) \\
 S1(L, x, \text{Left}(y, L', (z)L'')) &= \text{Left}(y, S1(L, x, L'), (z)S1(L, x, L'')), y \neq x \\
 S1(L, x, \text{Right}((y)L')) &= \text{Right}((y)S1(L, x, L')) \\
 S1(L, x, \text{Cut}(L', (y)L'')) &= \text{Cut}(S1(L, x, L'), (y)S1(L, x, L'')) \\
 S2(\text{Axiom}(y), x, L'_1, y', L'_2) &= \text{Left}(y, L'_1, y', L'_2) \\
 S2(\text{Left}(y, L_1, (z)L_2), x, L'_1, y', L'_2) &= \text{Left}(y, L_1, (z)S2(L_2, x, L'_1, y', L'_2)) \\
 S2(\text{Right}((y)L'), x, L'_1, y', L'_2) &= \text{Cut}(\text{Right}((y)L'), (x)\text{Left}(x, L'_1, y', L'_2)) \\
 S2(\text{Cut}(L_1, (y)L_2), x, L'_1, y', L'_2) &= \text{Cut}(L_1, (y)S2(L_2, x, L'_1, y', L'_2))
 \end{aligned}$$

Traditional assignment: The mapping $(-)^b : LJ \rightarrow \lambda$ is defined by

$$\begin{aligned}
 \text{Axiom}(x)^b &= x & \text{Right}((x)L)^b &= \lambda x.L^b \\
 \text{Left}(y, L_1, (x)L_2)^b &= [yL_1^b/x]L_2^b & \text{Cut}(L_1, (x)L_2)^b &= [L_1^b/x]L_2^b
 \end{aligned}$$

Under this mapping, some parts of cut-elimination are translated as β -reduction, but others receive no interpretation.

Proposition 10. *Let $R = \text{Log}$ (resp. $R \in \{S1, S2\}$). If $L_1 \rightarrow_R L_2$ in LJ , then $L_1^b \rightarrow_\beta L_2^b$ in λ (resp. $L_1^b = L_2^b$).*

A first attempt to overcome this situation is to consider the assignment $(-)^{\nabla} : LJ \rightarrow \lambda s$, which generates substitutions where $(-)^b$ calls meta-substitution:

$$\begin{aligned}
 \text{Axiom}(x)^{\nabla} &= x & \text{Right}((x)L)^{\nabla} &= \lambda x.L^{\nabla} \\
 \text{Left}(y, L_1, (x)L_2)^{\nabla} &= \langle yL_1^{\nabla}/x \rangle L_2^{\nabla} & \text{Cut}(L_1, (x)L_2)^{\nabla} &= \langle L_1^{\nabla}/x \rangle L_2^{\nabla}
 \end{aligned}$$

This mapping reveals a problem that was concealed by $(-)^b$.

“Imperfect substitution” in LJ : Let $L_0 = \text{Cut}(L_1, (x)\text{Left}(x, L_2, (y)L_3))$ and $L_4 = S1(L_1, x, \text{Left}(x, L_2, (y)L_3)) = \text{Cut}(L_1, x, \text{Left}(x, L'_2, (y)L'_3))$, where $L'_i = S1(L_1, x, L_i)$, $i = 2, 3$, and x is a free variable of L_2 or L_3 . Then $L_0 \rightarrow_{S1} L_4$. For

$i = 2, 3$, let $P_i = [L_1^b/x]L_i^b$. Then $L_0^b = [L_1^b/x][xL_2^b/y]L_3^b = [L_1^bP_2/y]P_3 = L_4^b$, the latter equality following by $P_i = L_i^b$, $i = 2, 3$.

Now consider $(-)^{\nabla}$ instead. Let $N_i = [L_1^{\nabla}/x]L_i^{\nabla}$, $i = 2, 3$. On the one hand $L_0^{\nabla} = \langle L_1^{\nabla}/x \rangle \langle xL_2^{\nabla}/y \rangle L_3^{\nabla} \rightarrow_{\sigma} [L_1^{\nabla}/x] \langle xL_2^{\nabla}/y \rangle L_3^{\nabla} = \langle L_1^{\nabla}N_2/y \rangle N_3 = M$, say. On the other hand $L_4^{\nabla} = \langle L_1^{\nabla}/x \rangle \langle xL_2^{\nabla}/y \rangle L_3^{\nabla}$. We would have liked $L_4^{\nabla} = M$, but L_4^{∇} is at least a σ -step behind: $L_4^{\nabla} \rightarrow \langle L_1^{\nabla}L_2^{\nabla}/y \rangle L_3^{\nabla}$. Unfortunately, these missing σ -steps propagate recursively, and we can expect $L_i^{\nabla} \rightarrow_{\sigma}^{\dagger} N_i$.

Why is L_4^{∇} a σ -step behind? Because, going back to L_4 , $S1$ is an imperfect substitution operator, which did not replace the free, head occurrence of x in $\text{Left}(x, L_2, (y)L_3)$ by L_1 . Instead, a cut is generated which $(-)^{\nabla}$ translates as the substitution $\langle L_1^{\nabla}/x \rangle -$. On the other hand, in $[L_1^{\nabla}/x] \langle xL_2^{\nabla}/y \rangle L_3^{\nabla}$ the free, head occurrence of x in $\langle xL_2^{\nabla}/y \rangle L_3^{\nabla}$ is indeed replaced by L_1^{∇} . These mismatches are not visible if meta-substitution is employed everywhere.

One way out is to extend LJ to a calculus where any term can enter the head position of $\text{Left}(-, L_2, (y)L_3)$ (see $\lambda\mathbf{s}$ later on). For now, we optimize $(-)^{\nabla}$ by performing the missing σ -steps at “compile time”.

Normalisation in $\lambda\mathbf{s}$ versus cut-elimination in LJ : We introduce mapping $(-)^{\nabla} : LJ \rightarrow \lambda\mathbf{s}$, which is defined exactly as $(-)^{\nabla}$, except for the clause for cuts, which now reads:

$$\begin{aligned} \text{Cut}(L_1, (x)\text{Left}(x, L_2, (y)L_3))^{\nabla} &= \langle L_1^{\nabla}L_2^{\nabla}/y \rangle L_3^{\nabla}, \text{ if } x \notin L_2, L_3 \\ \text{Cut}(L_1, (x)L_2)^{\nabla} &= \langle L_1^{\nabla}/x \rangle L_2^{\nabla}, \text{ if } \neg mll(x, L_2) \end{aligned}$$

Mapping $(-)^{\nabla}$ has better properties than mapping $(-)^{\nabla}$ as to preservation of reduction, but it introduces a typical identification. Suppose $x \notin L_2, L_3$ and let $L_0 = \text{Cut}(z, (x)\text{Left}(x, L_2, (y)L_3))$ and $L_4 = \text{Left}(z, L_2, (y)L_3)$. Notice that $L_0 \rightarrow_{S20} L_4$, and that L_0^{∇} and L_4^{∇} are the same $\lambda\mathbf{s}$ -term of the form $\langle zN_2/y \rangle N_3$.

We now obtain for $\lambda\mathbf{s}$ a result that improves Proposition [10](#). In order to achieve a good correspondence, we need to introduce in $\lambda\mathbf{s}$ an “eager” version of π , since the structural steps of cut-elimination in LJ perform *complete* permutations of cuts. First, we define certain contexts:

$$\mathcal{S} ::= \langle N/x \rangle [] \mid \langle N/x \rangle \mathcal{S}$$

Each \mathcal{S} is a $\lambda\mathbf{s}$ -term with a hole $[]$. $\mathcal{S}[P]$ denotes the result of filling P in the hole of \mathcal{S} . Next, “eager” π is defined by

$$(\pi') \quad \langle \mathcal{S}[V]N/x \rangle P \rightarrow \mathcal{S}[\langle V N/x \rangle P] ,$$

where V is a value. It is easy to show that each π' -step corresponds to a sequence of one or more π -steps.

Theorem 4 (Computational interpretation of cut-elimination). *Let $R \in \{S1, S21, S22, Log\}$. If $L_1 \rightarrow_R L_2$ (resp. $L_1 \rightarrow_{S20} L_2$) in LJ , then $L_1^{\nabla} \rightarrow_{\beta\pi\sigma}^{\dagger} L_2^{\nabla}$ in $\lambda\mathbf{s}$ (resp. $L_1^{\nabla} = L_2^{\nabla}$). In addition, $(-)^{\nabla}$ maps a reduction sequence ρ in LJ from L_1 to L_2 to a reduction sequence ρ^{∇} in $\lambda\mathbf{s}$ from L_1^{∇} to L_2^{∇} in a, so to say,*

structure-preserving way. To each R -step in ρ , there is a corresponding R' -steps in ρ^∇ , where R' is given according to the following table

	R	R'	computational interpretation
	$S1$	σ	execution of substitution
$S21 \cup S22$		π'	delaying of substitution
Log		β	generation of substitution

Moreover these R' -steps in ρ^∇ may be interleaved with trivial σ -steps of the form

$$\langle N_1/x \rangle \langle x N_2/y \rangle N_3 \rightarrow_\sigma \langle N_1 N_2/y \rangle N_3 \quad (x \notin N_2, N_3) . \quad (5)$$

The proof is postponed. It is useful to introduce here a new calculus $\lambda\mathbf{gs}$. By studying $\lambda\mathbf{gs}$, we will obtain a proof and two improvements of this theorem.

The $\lambda\mathbf{gs}$ -calculus: The $\lambda\mathbf{gs}$ -calculus is simultaneously an extension of $\lambda\mathbf{g}$ with a primitive substitution constructor, written $\langle N/x \rangle M$, and an extension of $\lambda\mathbf{s}$ where application is generalised. Reduction rules are as follows:

$$\begin{aligned}
 (\beta) \quad & (\lambda x.M)(N, y.P) \rightarrow \langle \langle N/x \rangle M / y \rangle P \\
 (\sigma) \quad & \langle N/x \rangle M \rightarrow [N/x]M \\
 (\pi_1) \quad & (\langle M/x \rangle N)(N', y.P') \rightarrow \langle M/x \rangle (N(N', y.P')) \\
 (\pi_2) \quad & \langle \langle M/x \rangle N / y \rangle P \rightarrow \langle M/x \rangle \langle N/y \rangle P \\
 (\pi_3) \quad & M(N, x.P)(N', y.P') \rightarrow M(N, x.P(N', y.P')) \\
 (\pi_4) \quad & \langle M(N, x.P) / y \rangle P' \rightarrow M(N, x.\langle P/y \rangle P') .
 \end{aligned}$$

Meta-substitution in $\lambda\mathbf{gs}$ is defined as expected. In particular, equation [\(II\)](#) holds again. Let $\pi = \cup_{i=0}^4 \pi_i$. A $\lambda\mathbf{gs}$ -term is in $\beta\pi\sigma$ -normal form iff it is in $\beta\pi_3\sigma$ -normal form iff it is a $\lambda\mathbf{g}$ -term in $\beta\pi$ -normal form iff it has no occurrences of substitution and every occurrence of generalised application in it is of the form $x(N, y, P)$.

Mappings between $\lambda\mathbf{gs}$ and LJ : There is an obvious injection of LJ into $\lambda\mathbf{gs}$. Formally, we define the mapping $(_)^\triangleright : LJ \rightarrow \lambda\mathbf{gs}$ as follows:

$$\begin{aligned}
 \text{Axiom}(x)^\triangleright &= x & \text{Right}((x)L)^\triangleright &= \lambda x.L^\triangleright \\
 \text{Left}(y, L_1, (x)L_2)^\triangleright &= y(L_1^\triangleright, x.L_2^\triangleright) & \text{Cut}(L_1, (x)L_2)^\triangleright &= \langle L_1^\triangleright / x \rangle L_2^\triangleright
 \end{aligned}$$

Hence, we may regard $\lambda\mathbf{gs}$ as an extension of LJ , where the particular case $y(N, x.P)$ of the generalised application construction plays the role of left introduction. Conversely, $M(N, x.P)$ may be regarded, when mapping back to LJ , as a primitive construction for a particular case of cut (except in the case of M being a variable). The mapping $(_)^\diamond : \lambda\mathbf{gs} \rightarrow LJ$ embodies this idea:

$$\begin{aligned}
 x^\diamond &= \text{Axiom}(x) & (\lambda x.M)^\diamond &= \text{Right}((x)M^\diamond) \\
 (y(N, x.P))^\diamond &= \text{Left}(y, N^\diamond, (x)P^\diamond) & (\langle N/x \rangle M)^\diamond &= \text{Cut}(N^\diamond, (x)M^\diamond) \\
 (M(N, x.P))^\diamond &= \text{Cut}(M^\diamond, (z)\text{Left}(z, N^\diamond, (x)P^\diamond)), & & \text{if } M \text{ is not a variable}
 \end{aligned}$$

with $z \notin N, P$ in the last equation.

Lemma 1. (1) For all $L \in LJ$: (i) $L^{\triangleright\circ} = L$. (ii) If L is cut-free, then L^{\triangleright} is $\beta\pi\sigma$ -normal. (2) For all $M \in \lambda\mathbf{gs}$, if M is $\beta\pi\sigma$ -normal, then M° is cut-free and $M^{\circ\triangleright} = M$.

Corollary 1. $(_)^\triangleright$ and $(_)^\circ$ are mutually inverse bijections between the sets of cut-free terms of LJ and the $\beta\pi\sigma$ -normal $\lambda\mathbf{gs}$ -terms²

If cuts are allowed, we cannot expect a bijective correspondence. Suppose $x \notin N_2, N_3$ and N_1 is not a variable. Let $M_1 = \langle N_1/x \rangle(x(N_2, y.N_3))$ and $M_2 = N_1(N_2, y.N_3)$. Then, M_1° and M_2° are the same term L , a cut of the form $\mathbf{Cut}(L_1, (x)\mathbf{Left}(x, L_2, (y)L_3))$. Notice that $M_1 \rightarrow_\sigma M_2$, by a *sigma*-step of the restricted form

$$\langle N_1/x \rangle(x(N_2, y.N_3)) \rightarrow_\sigma N_1(N_2, y.N_3) \quad (x \notin N_2, N_3). \quad (6)$$

Now, when mapping cut L back to $\lambda\mathbf{gs}$, there is, so to say, the M_1 option and the M_2 option. Mapping $(_)^\triangleright$ corresponds to the first option, whereas the second option corresponds to a refinement of $(_)^\triangleright$ named $(_)^\blacktriangleright$. This last mapping is defined exactly as $(_)^\triangleright$, except for the case of cuts, which now reads

$$\begin{aligned} \mathbf{Cut}(L_1, (x)\mathbf{Left}(x, L_2, (y)L_3))^\blacktriangleright &= L_1^\blacktriangleright(L_2^\blacktriangleright, y.L_3^\blacktriangleright), \text{ if } x \notin L_2, L_3 \\ \mathbf{Cut}(L_1, (x)L_2)^\blacktriangleright &= \langle L_1^\blacktriangleright/x \rangle L_2^\blacktriangleright, \text{ if } \neg \text{mll}(x, L_2) \end{aligned}$$

In particular, mappings $(_)^\triangleright$ and $(_)^\blacktriangleright$ coincide on cut-free terms.

Mapping $(_)^\blacktriangleright$ has better properties than mapping $(_)^\triangleright$ as to preservation of reduction, but it introduces a typical identification. Suppose $x \notin L_2, L_3$ and let $L_0 = \mathbf{Cut}(z, (x)\mathbf{Left}(x, L_2, (y)L_3))$ and $L_4 = \mathbf{Left}(z, L_2, (y)L_3)$. Then L_0^\blacktriangleright and L_4^\blacktriangleright are the same $\lambda\mathbf{gs}$ -term of the form $z(N_2, y.N_3)$. By the way, $L_0 \rightarrow_{S20} L_4$.

Normalisation in $\lambda\mathbf{gs}$ versus cut-elimination in LJ : We now investigate how mappings $(_)^\circ$ and $(_)^\blacktriangleright$ relate normalisation in $\lambda\mathbf{gs}$ and cut-elimination in LJ . To this end, we only need π_1 and π_3 among the π -rules of $\lambda\mathbf{gs}$. In addition, “eager” versions rules of π_1 and π_3 are required:

$$\begin{aligned} (\pi'_1) \quad & \langle M/x \rangle N(N', y.P') \rightarrow \langle M/x \rangle (N@'(N', y, P')) \\ (\pi'_3) \quad & M(N, x.P)(N', y.P') \rightarrow M(N, x.P@'(N', y, P')) \end{aligned} ,$$

where $M@'(N', y, P')$ is defined by recursion on M as follows: $x@'(N', y, P') = x(N', y.P')$; $(\lambda x.M)@'(N', y, P') = (\lambda x.M)(N', y.P')$; $(M(N, x.P))@'(N', y, P') = M(N, x.P@'(N', y, P'))$; and $(\langle M/x \rangle N)@'(N', y, P') = \langle M/x \rangle (N@'(N', y, P'))$.

Let $\pi' = \pi'_1 \cup \pi'_2$. It is easy to see that one π'_i -step ($i = 1, 3$) corresponds to one or more R -steps, where $R = \pi_1 \cup \pi_3$.

Proposition 11. Let $R \in \{\beta, \pi', \sigma\}$. If $M \rightarrow_R N$ in $\lambda\mathbf{gs}$, then $M^\circ \rightarrow_{S1, S2, Log}^+ N^\circ$ in LJ , except for some cases of $R = \sigma$, of the trivial form $\textcircled{0}$, for which one has $M^\circ = N^\circ$.

² It is well-known that cut-free LJ -terms are not in bijective correspondence with β -normal λ -terms; therefore, they are not in bijective correspondence with $\beta\pi\sigma$ -normal $\lambda\mathbf{s}$ -terms. In the particular case of $(_)^\blacktriangleright$, terms of the form $\mathbf{Left}(y, L_1, x.L_2)$ are always mapped to a substitution (which is a σ -redex).

Proof: By induction on $M \rightarrow_R N$, using the fact that, for all $M, N, P \in \lambda\mathbf{gs}$: (i) $S1(M^\circ, x, N^\circ) \rightarrow_{S20}^* [M/x]N^\circ$; (ii) if $x \notin N, P$ then $S2(M^\circ, x, N^\circ, y, P^\circ) = (M@N, y, P)^\circ$. ■

An inspection of the proof shows that $(-)^\circ$ maps a reduction sequence ρ in $\lambda\mathbf{gs}$ from M_1 to M_2 to a reduction sequence ρ° in LJ from M_1° to M_2° in a , so to say, structure-preserving way. Let $R \in \{\beta, \pi, \sigma\}$. To each R -step in ρ , there is a corresponding R' -step in ρ° , where R' is given by the left table below:

$$\begin{array}{c|c} R|R' & R|R' \\ \hline \sigma & S1 \cup S20 \\ \pi'_1 & S21 \cup S22 \\ \pi'_3 & S22 \\ \beta & Log \end{array} \qquad \begin{array}{c|c} R|R' & R|R' \\ \hline S1 & \sigma \\ S21 & \pi'_3 \\ S22 & \pi' \\ Log & \beta \end{array} \tag{7}$$

In addition, these R' -steps in ρ° may be interleaved with $S20$ -reduction steps.

Proposition 12. *Let $R \in \{S1, S21, S22, Log\}$. If $L_1 \rightarrow_R L_2$ (resp. $L_1 \rightarrow_{S20} L_2$) in LJ , then $L_1 \xrightarrow{\beta\pi\sigma}^+ L_2$ in $\lambda\mathbf{gs}$ (resp. $L_1 = L_2$). Moreover, a reduction sequence ρ in LJ from L_1 to L_2 to a reduction sequence ρ^\blacktriangleright in $\lambda\mathbf{gs}$ from L_1^\blacktriangleright to L_2^\blacktriangleright in a , so to say, structure-preserving way. To each R -step in ρ , there is a corresponding R' -step in ρ^\blacktriangleright , where R' is given according to the right table in (7). In addition, these R' -steps in ρ^\blacktriangleright may be interleaved with trivial σ -steps of the form (6).*

Proof: By induction on $L_1 \rightarrow_R L_2$ or $L_1 \rightarrow_{S20} L_2$. The proof uses the fact that, for all $L_1, L_2, L_3 \in LJ$: (i) either $\langle L_1^\blacktriangleright/x \rangle L_2^\blacktriangleright \rightarrow_\sigma \text{Cut}(L_1, (x)L_2)^\blacktriangleright$ or $\langle L_1^\blacktriangleright/x \rangle L_2^\blacktriangleright = \text{Cut}(L_1, (x)L_2)^\blacktriangleright$; (ii) $S1(L_1, x, L_2)^\blacktriangleright = [L_1^\blacktriangleright/x]L_2^\blacktriangleright$; (iii) if $x \notin L_2, L_3$ then $S2(L_1, x, L_2, y, L_3)^\blacktriangleright = L_1^\blacktriangleright@(L_2^\blacktriangleright, y, L_3^\blacktriangleright)$. An inspection of this inductive proof shows the statement regarding reduction sequences. ■

LJ versus $\lambda\mathbf{gs}$: Let us extract some lessons from the comparison between LJ and $\lambda\mathbf{gs}$ (where the latter is equipped with π' instead of π). The two systems are close. Cut-free terms and $\beta\pi\sigma$ -nfs are in bijective correspondence. Up to trivial reduction steps of the form $S20$ or (6), reduction sequences are similar, and obey a correspondence between reduction steps of the forms $S1, S2, Log$ and σ, π', β , respectively. However, $\lambda\mathbf{gs}$ is a preferable syntax for three reason: (1) it avoids the “imperfect substitution” problem; (2) it has a lighter notation; (3) reduction rules do not have side conditions. Side conditions in the reduction rules of LJ guarantee that a cut undergoes a $S2$ reduction only when it is not a $S1$ -redex. This sequencing is built in the syntax of $\lambda\mathbf{gs}$: a π -redex is never a σ -redex.

Mapping $\lambda\mathbf{gs}$ into $\lambda\mathbf{s}$: There is a bridge via $\lambda\mathbf{gs}$ between LJ and $\lambda\mathbf{s}$. We now close the bridge by studying mapping $(-)^* : \lambda\mathbf{gs} \rightarrow \lambda\mathbf{s}$. This mapping extends the one between $\lambda\mathbf{g}$ and $\lambda\mathbf{s}$, introduced in Section 3, with the clause $((N/x)M)^* = \langle N^*/x \rangle M^*$. As before with $\lambda\mathbf{g}$, mapping $(-)^*$ produces a strict simulation of reduction in $\lambda\mathbf{gs}$ by reduction in $\lambda\mathbf{s}$ (for the moment, π is taken in its “lazy” form both in $\lambda\mathbf{gs}$ and $\lambda\mathbf{s}$). So, the following is immediate.

Theorem 5 (SN). *If $M \in \lambda\mathbf{gs}$ is typable, then M is $\beta\pi\sigma$ -SN.*

Indeed, the simulation is perfect (one step in the source mapped to exactly one step in the target) for β and σ , and it is so for π if π is taken in the “eager” form. We introduce in $\lambda\mathbf{gs}$ an equivalent³ definition of rules π'_1 and π'_3 :

$$\begin{aligned} (\pi'_1) \quad & (\langle M/x \rangle \mathcal{C}[V])(N, y.P) \rightarrow \langle M/x \rangle \mathcal{C}[V(N, y, P)] \\ (\pi'_3) \quad & M(N, x.\mathcal{C}[V])(N, y.P) \rightarrow M(N, x.\mathcal{C}[V(N, y, P)]) \end{aligned} ,$$

where V is a value and \mathcal{C} is a context belonging to the class

$$\mathcal{C} ::= [] \mid M(N, x.\mathcal{C}) \mid \langle N/x \rangle \mathcal{C}$$

Proposition 13. *Let $R \in \{\beta, \sigma, \pi'\}$. If $M \rightarrow_R N$ in $\lambda\mathbf{gs}$, $M^* \rightarrow_R N^*$ in $\lambda\mathbf{s}$.*

Proof: Only π' remains to be checked. It is useful to define, for $M \in \lambda\mathbf{s}$, $\mathcal{S} = \langle M/x \rangle \mathcal{C}^*$ by recursion on \mathcal{C} as follows: $\langle M/x \rangle []^* = \langle M/x \rangle []$; $\langle M/x \rangle (P(N, y.\mathcal{C}))^* = \langle M/x \rangle (\langle P^*N^*/y \rangle \mathcal{C}^*)$; $\langle M/x \rangle (\langle N/y \rangle \mathcal{C})^* = \langle M/x \rangle (\langle N^*/y \rangle \mathcal{C}^*)$. By induction on \mathcal{C} one proves that, for all $M, N, P \in \lambda\mathbf{gs}$, $(\langle M/x \rangle \mathcal{C}[P])^* = (\langle M^*/x \rangle \mathcal{C}^*)[P^*]$ and $M(N, x.\mathcal{C}[P])^* = (\langle M^*N^*/x \rangle \mathcal{C}^*)[P^*]$. Next we do an induction on $M \rightarrow_{\pi'} N$. ■

We can finally give a proof of Theorem 4. It follows from Propositions 12 and 13, and the fact that $(-)^{\blacktriangledown} = (-)^* \circ (-)^{\blacktriangleright}$ and that $(-)^*$ maps reduction steps of the form (6) to reduction steps of the form (5).

We finish by obtaining strong cut-elimination⁴ for LJ as a corollary to strong normalisation for $\lambda\mathbf{s}$. Indeed, all we need is Theorem 4, together with the fact that \rightarrow_{S20} is terminating and that $(-)^{\blacktriangledown}$ preserves typability.

Theorem 6 (Strong cut-elimination). *Let $R = S1 \cup S2 \cup \text{Log}$. For all $L \in LJ$, if L is typable, then L is R -SN.*

Improving the computational interpretation: Theorem 4 is an improvement of Proposition 10 as to the interpretation of cut-elimination. Depending on the role we attribute to $\lambda\mathbf{gs}$, we can see Propositions 12 and 13 as improvements over Theorem 4 w.r.t. the same goal. On the one hand, if we regard $\lambda\mathbf{gs}$ as an adaptation of $\lambda\mathbf{s}$ particularly suited for the comparison with LJ , then Proposition 12 improves Theorem 4, because it includes a bijection between cut-free terms and $\beta\sigma\pi$ -normal forms. On the other hand, if we regard $\lambda\mathbf{gs}$ as an adaptation of LJ which avoids the problem of “imperfect substitution”, then Proposition 13 improves Theorem 4, because it states a perfect correspondence that avoids the little perturbations of having $S20$ -steps in the source reduction sequence that are not simulated, or extra σ -steps interleaved in the target reduction sequence.

³ The equivalence follows from two facts: (1) for all $V, N, P \in \lambda\mathbf{gs}$, $\mathcal{C}[V]@(N, y, P) = \mathcal{C}[V(N, y, P)]$; (2) every $M \in \lambda\mathbf{gs}$ can be written as $\mathcal{C}[V]$, with V value.

⁴ A (weak) cut-elimination result is obtained as follows. Let $L \in LJ$. From $M = L^{\blacktriangleright}$, one gets a $\beta\pi\sigma$ -nf N . Proposition 11 guarantees that $L = L^{\blacktriangleright\blacktriangleright} \rightarrow^* N^\circ$ in LJ . Since N is a $\beta\pi\sigma$ -nf, N° is cut-free.

However, in Proposition [13](#) the mismatch between $\beta\sigma\pi$ -normal forms in λ_{gs} and in λ_{s} remains, and in Proposition [12](#) the little perturbations related to $S20$ -steps and extra σ -steps survive (Theorem [4](#) shared both defects). For an *isomorphism* between sequent calculus and natural deduction, see [3](#).

References

1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. *Journal of Functional Programming* 1(4), 375–416 (1991)
 2. Danos, V., Joinet, J.-B., Schellinx, H.: A new deconstructive logic: linear logic. *The Journal of Symbolic Logic* 62(2), 755–807 (1997)
 3. Espírito Santo, J.: Completing Herbelin’s programme. In: *Proceedings of TLCA’07*. LNCS, Springer, Heidelberg (2007)
 4. Gallier, J.: Constructive logics. Part I: A tutorial on proof systems and typed λ -calculi. *Theoretical Computer Science* 110, 248–339 (1993)
 5. Herbelin, H.: A λ -calculus structure isomorphic to a Gentzen-style sequent calculus structure. In: Pacholski, L., Tiuryn, J. (eds.) *CSL 1994*. LNCS, vol. 933, pp. 61–75. Springer, Heidelberg (1995)
 6. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel’s T. *Archive for Mathematical Logic* 42, 59–87 (2003)
 7. Joachimski, F.: On Zucker’s isomorphism for LJ and its extension to pure type systems (Submitted 2003)
 8. Moggi, E.: Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh (1988)
 9. Ohta, Y., Hasegawa, M.: A terminating and confluent linear lambda calculus. In: Pfenning, F. (ed.) *RTA 2006*. LNCS, vol. 4098, pp. 166–180. Springer, Heidelberg (2006)
 10. Regnier, L.: Une équivalence sur les lambda-termes. *Theoretical Computer Science* 126(2), 281–292 (1994)
 11. Revesz, G.: *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge Tracts in Theoretical Computer Science, vol. 4. Cambridge University Press, Cambridge (1988)
 12. Rose, K.: Explicit substitutions: Tutorial & survey. Technical Report LS-96-3, BRICS (1996)
 13. van Raamsdonk, F., Severi, P., Sorensen, M., Xi, H.: Perceptual reductions in λ -calculus. *Information and Computation* 149(2), 173–225 (1999)
 14. Vestergaard, R., Wells, J.: Cut rules and explicit substitutions. In: *Second International Workshop on Explicit Substitutions* (1999)
 15. von Plato, J.: Natural deduction with general elimination rules. *Annals of Mathematical Logic* 40(7), 541–567 (2001)
- [On Theories with a Combinatorial Definition of “Equivalence”](#)

Innermost-Reachability and Innermost-Joinability Are Decidable for Shallow Term Rewrite Systems*

Guillem Godoy¹ and Eduard Huntingford²

¹ Technical University of Catalonia
Jordi Girona 1, Barcelona, Spain
ggodoy@lsi.upc.edu

² Technical University of Catalonia
Jordi Girona 1, Barcelona, Spain
eduard.hl@gmail.com

Abstract. Reachability and joinability are central properties of term rewriting. Unfortunately they are undecidable in general, and even for some restricted classes of term rewrite systems, like shallow term rewrite systems (where variables are only allowed to occur at depth 0 or 1 in the terms of the rules).

Innermost rewriting is one of the most studied and used strategies for rewriting, since it corresponds to the "call by value" computation of programming languages. Henceforth, it is meaningful to study whether reachability and joinability are indeed decidable for a significant class of term rewrite systems with the use of the innermost strategy.

In this paper we show that reachability and joinability are decidable for shallow term rewrite systems assuming that the innermost strategy is used. All of these results are obtained via the definition of the concept of weak normal form, and a construction of a finite representation of all weak normal forms reachable from every constant. For the particular left-linear shallow case and assuming that the maximum arity of the signature is a constant, these results are obtained with polynomial time complexity.

1 Introduction

Reachability and joinability are central properties of term rewrite systems (TRS). A term t is reachable from a term s if there exists a rewrite sequence that transforms s into t . Two terms u and v are joinable if there exists a term w reachable from u and v .

Reachability and joinability are undecidable in general, and even if we restrict to linear TRS (variables at every side of a rule occur at most once), or to shallow

* The both authors were supported by Spanish Min. of Educ. and Science by the LogicTools project (TIN2004-03382). The second author was also supported by Spanish Min. of Educ. and Science by the GRAMMARS project (TIN2004-07925-C03-01).

TRS [9] (variables occur at depth 0 or 1 in the terms of every rule). The more powerful positive results for decidability are based on the property of effective preservation of regularity. A TRS is effectively regularity preserving when the set of terms reachable from a given regular set (a set of terms recognizable by a tree automaton [1]) is also regular, and computable from the initial regular set. For this and other reasons, effective preservation of regularity has been extensively studied [11,2,8,10,12,13]. These results include in particular all TRS that are right-shallow (variables in the right terms of the rules occur at depth 0 or 1) and right-linear (variables in the right terms of the rules occur at most once).

In several cases rewriting is considered restricted to the particular strategy of innermost rewriting, that corresponds to the *call by value* computation of programming languages, where arguments are fully evaluated before the function application. Henceforth, it is meaningful to study whether reachability and joinability are indeed decidable for a significant class of TRS under the assumption of the use of the innermost strategy.

At first look deciding properties like reachability and joinability seems to be more difficult for innermost rewriting, since we have not such powerful regularity preserving results. But surprisingly we can deal with shallow TRS, which are not regularity preserving even for general rewriting.

In this paper, we prove the decidability of reachability and joinability for shallow TRS assuming the use of the innermost strategy. All of these results are based on an initial common construction. We define the concept of *weak normal form* as any term such that all its subterms at depth 1 are either constants or normal forms. After that we compute a finite representation of all weak normal forms reachable from every constant. This representation that we call *WNF-representation* is based on a kind of constrained terms. Its usefulness relies on the fact that in any innermost derivation with a shallow TRS, one gets a weak normal form just after every rewrite step at the top. Finally, we prove decidable the problems of reachability and joinability making an adequate use of the WNF-representation. For the particular case of left-linear shallow TRS and assuming that the maximum arity of the signature is a constant these results are obtained with polynomial time complexity.

The paper is structured as follows. In Section 2 we introduce some basic notions and notations. In Section 3 we present transformations that replace the shallow rules by flat rules, and allow us to simplify the arguments in the rest of the paper. In Section 4 we define weak normal forms and a kind of constrained terms that represent weak normal forms. Before dealing with shallow TRS, in Section 5 we show how to compute the WNF-representation for left-linear shallow TRS for two reasons. On the one side for explanation purposes, since this simpler case allows to understand better the later general construction. On the other side, this case is interesting due to its time complexity, as we have mentioned before. In Section 6 we show in two parts how the WNF-representation can be computed for shallow TRS. In the first part we give a (possibly non-terminating) process that computes this finite representation. In the second part we obtain

a computable bound for execution time of the process necessary to complete the finite representation. In Sections 7 and 8 we decide the reachability and joinability problems making use of the WNF-representation. Finally, in Section 9 we deal with complexity issues.

2 Preliminaries

We use standard notation from the term rewriting literature. A signature Σ is a (finite) set of function symbols, which is partitioned as $\cup_i \Sigma_i$ such that $f \in \Sigma_n$ if the arity of f is n . Symbols in Σ_0 , called *constants*, are denoted by a, b, c, d, e , with possible subscripts. The elements of a set \mathcal{V} of variable symbols are denoted by x, y, z with possible subscripts. The set $\mathcal{T}(\Sigma, \mathcal{V})$ of *terms* over Σ and \mathcal{V} , *position* p in a term, *subterm* $t|_p$ of term t at position p , and the term $t[s]_p$ obtained by replacing $t|_p$ by s are defined in the standard way. For example, if t is $f(a, g(b, h(c)), d)$, then $t|_{2.2.1} = c$, and $t[d]_{2.2} = f(a, g(b, d), d)$. The empty sequence, denoted by λ , corresponds to the root position. We write $p_1 > p_2$ (equivalently, $p_2 < p_1$) and say p_1 is below p_2 (equivalently, p_2 is above p_1) if p_2 is a proper prefix of p_1 , that is, $p_1 = p_2.p'_2$ for some nonempty p'_2 . Positions p and q are *disjoint* if $p \not\preceq q$ and $q \not\preceq p$. By $Vars(t)$ we denote the set of all variables occurring in t . The *height* of a term s is 0 if s is a variable or a constant, and $1 + \max_i height(s_i)$ if $s = f(s_1, \dots, s_m)$. The *depth* of an occurrence at position p of a term t in a term s , i.e. $s = s[t]_p$, is $|p|$. The *size* of a term $s = f s_1 \dots s_m$, denoted by $|s|$, is $1 + \sum_{i=1}^m size(s_i)$. Usually we will denote a term $f(t_1, \dots, t_n)$ by the simplified form $f t_1 \dots t_n$, and $t[s]_p$ by $t[s]$ when p is clear by the context or not important.

A substitution σ is sometimes presented explicitly as $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$. We assume standard definitions for a *rewrite rule* $l \rightarrow r$, a *term rewrite system* R , the *one step rewrite relation at position p induced by R* $\rightarrow_{R,p}$, and the *one step rewrite relation induced by R* (at any position) \rightarrow_R (denoted also as \rightarrow if R is clear by the context). A rewrite step at a position different from λ is sometimes denoted by $s \rightarrow_{R, > \lambda} t$. We do the usual assumptions for the rules $l \rightarrow r$ of a TRS R , i.e. l is not a variable, and all variables occurring in r also occur in l .

The notations \leftrightarrow , \rightarrow^+ , and \rightarrow^* , are standard [3]. R is terminating if no infinite derivation $s_1 \rightarrow_R s_2 \rightarrow \dots$ exists. A term t is *reachable* from s by R (or, R -reachable) if $s \rightarrow_R^* t$. A term t is *reachable* from a set of terms S by R if t is reachable from all terms in S . A term s is *R -irreducible* (or, in R -normal form) if there is no term t such that $s \rightarrow_R t$. Two terms s and t are *R -joinable* if there exists a term u reachable from s and t . A set S of terms is *R -joinable* if there exists a term u reachable from all terms in S . A (*rewrite*) *derivation* (from s) is a sequence of rewrite steps (starting from s), that is, a sequence $s \rightarrow_R s_1 \rightarrow_R s_2 \rightarrow_R \dots$. With $s \rightarrow_R^* t$ we will denote that t is R -reachable from s , or a concrete derivation from s to t , depending on the context. The length of a derivation $s \rightarrow_R^* t$, denoted as $|s \rightarrow_R^* t|$, is the number of steps of the derivation. Depending on the context, $|s \rightarrow_R^* t|$ denotes the length of the minimum derivation $s \rightarrow_R^* t$.

A term t is called *ground* if t contains no variables. It is called *shallow* if all variable positions in t are at depth 0 or 1. It is *flat* if its height is at most 1. A rule $l \rightarrow r \in R$ is called *shallow* if both l, r are shallow; it is *flat* if both l, r are flat.

A rewrite step $s = s[u]_p \rightarrow_{R,p} s[v]_p$ is innermost if all proper subterms of u are normal forms. In the rest of the article we will always assume that all rewrite steps are innermost, and that the problems we deal with, like reachability and joinability, refer to innermost rewriting. In particular, whenever we say that a term s reaches a term t we mean that s innermost reach t .

3 Simplifying Assumptions

In this section we comment some simplifying transformations on the signature and the given shallow TRS. Similar transformations have appeared in other previous works [6,5], and we just comment them.

We will always assume that all terms are constructed over a given fixed signature Σ that contains several constants and only one non-constant function symbol f . If this was not the case, we can define a transformation T from terms over Σ into terms over a new signature Σ' as follows. Let m be the maximum arity of a symbol in Σ plus 1. We chose a new function symbol f with arity m and define the new signature $\Sigma' = \Sigma'_0 \cup \Sigma'_m$ as $\Sigma'_0 = \Sigma$ and $\Sigma'_m = \{f\}$. Note that all symbols of Σ appear also in Σ' but with arity 0. Now, we recursively define $T : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma', \mathcal{V})$ as $T(c) = c$ and $T(x) = x$ for constants $c \in \Sigma_0$ and variables $x \in \mathcal{V}$, and $T(gt_1 \dots t_k) = f(T(t_1), \dots, T(t_k), g, \dots, g)$ for terms headed with $g \in \Sigma - \Sigma_0$.

Lemma 1. *Let s, t be terms in $\mathcal{T}(\Sigma, \mathcal{V})$ and R a shallow TRS with all its terms in $\mathcal{T}(\Sigma, \mathcal{V})$. Let $T(R)$ be $\{T(l) \rightarrow T(r) \mid l \rightarrow r \in R\}$.*

Then, $T(R)$ is shallow, s innermost- R -reaches t iff $T(s)$ innermost- $T(R)$ -reaches $T(t)$, and s and t are innermost- R -joinable iff $T(s)$ and $T(t)$ are innermost- $T(R)$ -joinable.

Finally, we will also assume that all rules in R are flat. If this was not the case, we can modify R as follows. First remove all rules $l \rightarrow r$ such that l has a proper subterm that is not a normal form: note that in such a case this rule is useless for innermost rewriting. Second, we proceed by applying the following transformations whenever is possible.

- a) If there is a non-constant normal form t that is a proper subterm of a left-hand-side of a rule in R , then create a new constant c , replace all occurrences of t in the left-hand-sides of the rules of R by c , and add the rule $t \rightarrow c$ to R .
- b) If there is a non-constant term t that is a proper subterm of a right-hand-side of a rule in R , then create a new constant c , replace all occurrences of t in the right-hand-sides of the rules of R by c , and add the rule $c \rightarrow t$ to R .

At every step decreases the total sum of the number of positions at depth more than one in all of the left and right hand sides of R . Hence, this process terminates.

Lemma 2. *Let s, t be terms in $\mathcal{T}(\Sigma, \mathcal{V})$ and R a shallow TRS with all its terms in $\mathcal{T}(\Sigma, \mathcal{V})$. Let R^∞ be the resulting TRS of the above transformation.*

Then R^∞ is flat, s innermost- R -reaches t iff s innermost- R^∞ -reaches t , and s and t are innermost- R -joinable iff s and t are innermost- R^∞ -joinable.

4 Weak Normal Forms and Constrained Terms

Definition 1. *A term t is a weak normal form if it is either a constant or a non-constant term of the form $t = ft_1 \dots t_m$ such that every t_i is either a constant or a normal form.*

We will write $\mathcal{P}(S)$ to denote the powerset of S minus the empty set.

Definition 2. *A constraint C is a partial mapping $C : \mathcal{V} \rightarrow \mathcal{P}(\Sigma_0)$, i.e. an assignment from variables to non-empty sets of constants.*

Given a constraint C , a TRS R , and a substitution σ , we say that σ is a solution of C (w.r.t. R) if for all x in $\text{Dom}(C)$ it holds that $x\sigma$ is a normal form w.r.t. R , $x\sigma$ is reachable from $C(x)$, and $x\sigma$ is not in Σ_0 .

A constrained term is a pair (t, C) , where t is a flat term and C is a constraint, defined only for $\text{Vars}(t)$, which we will denote as $t|C$. A term $t\sigma$ is an instance of $t|C$ if σ is a solution of C .

Note that then any instance of a constrained term is a weak normal form.

5 WNF-Representation for Flat Left-Linear TRS

We describe an algorithm that computes, for every constant c , a set of constrained terms r_c , and an auxiliary boolean value n_c . Its goal is make r_c to contain a set of constrained terms such that their instances are all the weak normal forms reachable from c , and make n_c to be true if and only if there exists some non-constant normal form reachable from c . The basic idea is to compute many rewrite steps at once, by anticipating them. Set (a) deals with rewrite steps at λ , set (b) anticipates several rewrite steps at depth 1 that transform a constant into an other constant, and set (c) anticipates the transformation of a constant at depth 1 into a non-constant normal form.

WNF-Representation Algorithm for flat left-linear TRS

1. For every constant c do $r_c := \{c|\emptyset\}$, and $n_c := \text{False}$.
2. For every constant c do:

$$\begin{aligned}
r_c &:= r_c \cup \\
\text{(a) } \{t|C'\} & \quad \left| \begin{array}{l} \exists (s|C) \in r_c : (s \rightarrow_{R,\lambda} t \wedge C' \subseteq C) \wedge \\ \text{Dom}(C') = \text{Vars}(t) \end{array} \right. \cup \\
\text{(b) } \{t[e]_j|C\} & \quad \left| \begin{array}{l} (t|C) \in r_c \wedge t|_j \in \Sigma_0 \wedge e \in \Sigma_0 \wedge \\ (e|\emptyset) \in r_{t|_j} \end{array} \right. \cup \\
\text{(c) } \{t[x]_j|(C \cup \{(x, \{t|_j\})\})\} & \quad \left| \begin{array}{l} (t|C) \in r_c \wedge t|_j \in \Sigma_0 \wedge x \notin \text{Dom}(C) \wedge \\ n_{t|_j} = \text{True} \end{array} \right. \cup
\end{aligned}$$

$$n_c := n_c \vee \exists (t|C) \in r_c : t \text{ is a non-constant normal form}$$

3. If some r_c or n_c has changed in the previous step, up to renaming of variables, then go to 2.

Note that at every execution step the sets r_c can only increase, and that the values n_c can only pass from **False** to **True**. Note also that the generated constrained terms are always of the form $c|\emptyset$ or $x|\{(x, \{c\})\}$ or $f\alpha_1 \dots \alpha_m|C$, where the α_i 's are constants or variables and C is a set of pairs of the form $(\alpha_i, \{c\})$, just one for every different variable α_i . If n is the total number of constants, then at most $n + n + (n + n + m)^m$ of these constrained terms exists, up to renaming of variables. Since every execution step does simple operations of matching on these data, the total cost of the algorithm is $(n + m)^{\mathcal{O}(m)}$.

We have not indicated the order in which the assignments $r_c := \dots$ and $n_c := \dots$ for the different c 's are executed. This is not important for the goal of the process, but, for simplicity purposes in later arguments, we will assume that all of them are computed at once, as a multiple assignment.

We will denote with r_c^∞ and n_c^∞ the limit of the values of the corresponding variables r_c and n_c in the WNF-representation algorithm.

Theorem 1. (*correctness*) *For every c and every constrained term $(t|C)$ in r_c^∞ , there exists at least one instance of $(t|C)$, and all instances of $(t|C)$ are innermost-reachable from c .*

For every c , if $n_c^\infty = \text{True}$, then there exists a non-constant normal t form innermost-reachable from c .

Proof. We prove both facts for all r_c^i and n_c^i by induction on the number of steps i in the execution process. In step 1, after the instruction $r_c := \{c|\emptyset\}$ the property is satisfied since c is reachable from c , which is the only instance of $c|\emptyset$. Let us analyze step 2, and suppose that every r_c and n_c satisfies the assumption before one of the assignments $r_c := r_c \cup \dots$. In this instruction several elements are added to r_c , and we will show that all of them satisfy the assumption.

- Case $\{t|C'\} \mid \exists (s|C) \in r_c : (s \rightarrow_{R,\lambda} t \wedge C' \subseteq C \wedge \text{Dom}(C') = \text{Vars}(t))$. By induction hypothesis $s|C$ has some instance $s\sigma$. Since $C' \subseteq C$, σ is a solution of C' and hence, $t\sigma$ is an instance of $t|C'$. Now, let $t\sigma'$ be any instance of $t|C'$. W.l.o.g. we assume σ' defined only for $\text{Dom}(C')$. Using the previous σ ,

we extend σ' to the variables x of $\mathcal{D}om(C) - \mathcal{D}om(C')$ as $\sigma'(x) = \sigma(x)$. Now, σ' satisfies also C , and $s\sigma'$ is an instance of $s|C$. By induction hypothesis $s\sigma'$ is reachable from c . Since the rewrite step $s \rightarrow_{R,\lambda} t$ is innermost and σ' is a replacement of variables by normal forms, the rewrite step $s\sigma' \rightarrow_{R,\lambda} t\sigma'$ using the same rule is also innermost. Therefore $t\sigma'$ is reachable from c .

- Case $\{t[e]_j|C \mid (t|C) \in r_c \wedge t|_j \in \Sigma_0 \wedge e \in \Sigma_0 \wedge (e|\emptyset) \in r_{t|_j}\}$. By induction hypothesis, there exists an instance $t\theta$ of $t|C$, and hence θ is a solution of C , and $t[e]_j\theta$ is an instance of $t[e]_j|C$.

For any instance $t[e]_j\sigma$ of $t[e]_j|C$, we have the instance $t\sigma$ for $t|C$, which is reachable from c by induction hypothesis. Similarly, e is reachable from $t|_j$ by induction hypothesis. We conclude that $t[e]_j\sigma$ is reachable from c .

- Case $\{t[x]_j|(C \cup \{(x, \{t|_j\})\}) \mid (t|C) \in r_c \wedge t|_j \in \Sigma_0 \wedge x \notin \mathcal{D}om(C) \wedge n_{t|_j} = \mathbf{True}\}$. By induction hypothesis, there exists an instance $t\theta$ of $t|C$. W.l.o.g. we assume θ defined only for $\mathcal{D}om(C)$. As $n_{t|_j} = \mathbf{True}$ there is a non-constant normal form s that is reachable from $t|_j$ by induction hypothesis. If we extend θ to x such that $x\theta = s$, then $t[x]_j\theta$ is an instance of $t[x]_j|(C \cup \{(x, \{t|_j\})\})$. For any instance $t[x]_j\sigma$ of $t[x]_j|(C \cup \{(x, \{t|_j\})\})$, it holds that $x\sigma$ is a non-constant normal form reachable from $\{t|_j\}$. Moreover, $t\sigma$ is an instance of $t|C$, which is reachable from c by induction hypothesis. We conclude that $t[x]_j(\sigma \cup (x, s))$ is reachable from c .

Let us analyze the following assignment when it makes n_c to change from **False** to **True**.

$n_c := n_c \vee \exists (t|C) \in r_d : t$ is a non-constant normal form

By the condition, t is a non-constant normal form. Since $t|C \in r_c$, by induction hypothesis there exists an instance $t\sigma$ of t , and $t\sigma$ is reachable from c . To conclude it is enough to see that $t\sigma$ is in fact a normal form. By the definition of instance of a constrained term we have that $t\sigma$ is a weak normal form. Hence, all its non-constant subterms at depth 1 are normal forms. Moreover, since t is a normal form, all the constant subterms at depth 1 are also normal forms. But $t\sigma$ neither can be rewritten at position λ , since t can not and R is left-linear. \square

(Recall that \rightarrow^* is understood as innermost rewriting with R .)

Theorem 2. (completeness) *For every constant c and every weak normal form s innermost-reachable from c , there exists some constrained term $t|C$ in r_c^∞ such that s is an instance $t\sigma$ of $t|C$, and for all $(x, \{d\})$ in C it holds that $|d \rightarrow^* x\sigma| < |c \rightarrow^* s|$.*

*Moreover, if there exists a non-constant normal form s innermost-reachable from c , then n_c^∞ is **True**.*

Proof. We prove both facts by induction on $|c \rightarrow^* s|$, but considering the first fact smaller: when proving the second fact we will assume that the first fact is true for smaller than or equal values, and that the second fact is true for smaller values.

Since $r_c^\infty = \bigcup r_c^i$ and $n_c^\infty = \bigvee n_c^i$ for every c , it is enough to see that every of such $t|C$ and **True** have been added or assigned to the corresponding r_c

and n_c , respectively, at some point of the execution of the WNF-representation algorithm.

Fact 1: Let s be a weak normal form reachable from a certain c . We consider a derivation $c \rightarrow^* s$ such that $|c \rightarrow^* s|$ is minimum. If $c \rightarrow^0 s$, then $s = c$, and $c|\emptyset$ has been added to r_c in the first step of the WNF-representation algorithm. Otherwise, the derivation $c \rightarrow^* s$ has at least one step at λ . We decompose this derivation considering the last of such steps at λ as $c \rightarrow^* u \rightarrow_{l \rightarrow r, \lambda} v \rightarrow^* s$.

Note that u and v are weak normal forms, and since $c \rightarrow^* s$ was chosen minimal, $|c \rightarrow^* u| < |c \rightarrow^* s|$. By induction hypothesis, there exists $u'|C$ in r_c^∞ such that u is an instance $u'\sigma$ of $u'|C$, and for all $(x, \{d\})$ in C it holds that $|d \rightarrow^* x\sigma| < |c \rightarrow^* u| < |c \rightarrow^* s|$.

Note that, if $u|_j$ is a constant, then $u'|_j = u|_j$. Therefore, since R is linear, the rule $l \rightarrow r$ is also applicable on u' at position λ . Let v' be such that $u' \rightarrow_{l \rightarrow r, \lambda} v'$. Instantiating previous rewrite step with σ we obtain $u'\sigma \rightarrow_{l \rightarrow r, \lambda} v'\sigma$. Since $u'\sigma$ is u , $v'\sigma$ has to be v . Let C' be such that $C' \subseteq C$ and $\text{Dom}(C') = \text{Vars}(v')$. Then, σ is a solution of C' and $v'\sigma = v$ is an instance of $v'|C'$. Moreover, the constrained term $v'|C'$ has been added to r_c due to set (a) of the assignment in step 2. of the WNF-representation algorithm.

If v' is a constant, then $v' = v = s$, since there are no rewrite steps at λ in $v \rightarrow^* s$. In this case C' is empty and we are done. If v' is a variable, then v is a normal form and hence $v = s$. In this case $\text{Dom}(C') \subseteq \text{Dom}(C)$ and we are done. Hence, assume that v' is height 1, v is of the form $fv_1 \dots v_m$, s is of the form $fs_1 \dots s_m$, and we have derivations $v_j \rightarrow^* s_j$ with less number of steps than $c \rightarrow^* s$. Recall that s is a weak normal form, and hence, the s_i 's are either constants or normal forms. If v_j is a non-constant term, then it is a normal form and $v_j = s_j$. If v_j is a constant term then either s_j is a constant that belongs to $r_{v_j}^\infty$ by induction hypothesis, or a normal form and $n_{v_j}^\infty$ is true, again by induction hypothesis. We define s' by replacing in v' , every constant v_j by either s_j when s_j is a constant, or by a new variable x_j when s_j is not a constant. We also define $D = C' \cup \{(x_j, \{v_j\}) \mid v_j \in \Sigma_0, s_j \notin \Sigma_0\}$, and $\gamma = \sigma \cup \{x_j \mapsto s_j \mid v_j \in \Sigma_0, s_j \notin \Sigma_0\}$. Due to sets (b) and (c) of the assignment in step 2 and by repeated executions for every of such v_j 's, the constrained term $s'|D$ has been added to r_c and has $s = s'\gamma$ as instance. Moreover, for all $(x, \{d\})$ in D it holds that $|d \rightarrow^* x\sigma| < |c \rightarrow^* s|$.

Fact 2: Let s be a non-constant normal form reachable from c . We must show that n_c^∞ is true. By induction hypothesis there exists a constrained term $t|C$ in r_c^∞ such that s is an instance $t\sigma$ of $t|C$. Due to the assignment $n_c := \dots$ in the WNF-representation algorithm to conclude it suffices to see that t is a non-constant normal form. If t is a variable then it is a non-constant normal form. Otherwise it is of the form $ft_1 \dots t_m$. If $s|_j$ is a constant, then $s|_j = t_j$. Therefore, by the left-linearity of R , any rule applicable on t at position λ is also applicable on s at position λ , and hence, such a rule does not exist. Hence, t is a normal form. \square

6 WNF-Representation for Flat TRS

6.1 A (non-terminating) Process for Computing a WNF-Representation

We describe a (not necessarily terminating) process that computes, for every constant c , two sets of constrained terms r_c and \bar{r}_c , and for every set $S \in \mathcal{P}(\Sigma_0)$, a set of non-constant normal forms. Its goal is make r_c to contain a set of constrained terms such that their instances are all the weak normal forms reachable from c , make \bar{r}_c to contain a set of flat constrained terms such that their normal form instances are all the non-constant normal forms reachable from c , and make n_S to contain all the non-constant normal forms reachable from S (that could be infinite).

(In order to simplify notation, in some cases we will write $\exists(t|C) \in r_c$ instead of $\exists C : (t|C) \in r_c$ or $\exists t : (t|C) \in r_c$, depending on the case.)

WNF-representation Process

1. For every constant c do $r_c := \{c|\emptyset\}$, $\bar{r}_c := \emptyset$, and for every $S \in \mathcal{P}(\Sigma_0)$ do $n_S := \emptyset$.

2. For every constant c do:

$r_c := r_c \cup$

(a) $\{t|C'\}$ $\mid \exists(s|C) \in r_c : (s \rightarrow_{R,\lambda} t \wedge C' \subseteq C \wedge \text{Dom}(C') = \text{Vars}(t)) \cup$

(b) $\{t|D\}$ $\mid \exists(t|C) \in r_c : \forall x : (C(x) \subseteq D(x) \wedge n_{D(x)} \neq \emptyset) \cup$

(c) $\{t\sigma|C\sigma\}$ $\mid \sigma$ is a substitution-to-variables, and $(t|C) \in r_c$ and $\forall x, y : (x\sigma = y\sigma \Rightarrow C(x) = C(y)) \cup$

(d) $\{t[e]_j|C\}$ $\mid (t|C) \in r_c \wedge t|_j \in \Sigma_0 \wedge e \in \Sigma_0 \wedge (e|\emptyset) \in r_{t|_j} \cup$

(e) $\{t[x]_j|(C \cup \{(x, \{t|_j\})\})\}$ $\mid (t|C) \in r_c \wedge t|_j \in \Sigma_0 \wedge x \notin \text{Dom}(C) \wedge n_{\{t|_j\}} \neq \emptyset$

$\bar{r}_c := \bar{r}_c \cup \{t|C \mid t|C \in r_c \text{ and the height of } t \text{ is } 1\} \cup$

$\{ft_1 \dots t_m|C \mid \exists x \{(x, S)\} \in r_c : \forall d \in S :$

$ft_1 \dots t_m|C \in \bar{r}_d \text{ up to renaming}\} \cup$

$\{ft_1 \dots t_m|D \mid \exists(ft_1 \dots t_m|C) \in \bar{r}_c : \forall x : (C(x) \subseteq D(x) \wedge n_{D(x)} \neq \emptyset)\}$

3. If some r_c or \bar{r}_c has changed in previous step, up to renaming of variables, then go to 2.

4. For every $S \in \mathcal{P}(\Sigma_0)$ do:

$$n_S := n_S \cup \{t\sigma \mid \exists C : \forall d \in S : (t|C) \in \bar{r}_d \text{ up to renaming, and} \\ \forall x \in \text{Dom}(C) : (x\sigma \in n_{C(x)}) \text{ and} \\ t\sigma \text{ is a non-constant normal form} \}$$

5. If some n_S has changed in the previous step, go to 2.

As for the WNF-Representation Algorithm for flat left-linear TRS, we will assume that the assignments $r_c := \dots, \bar{r}_c := \dots$ for the different c 's are executed at once, as a multiple assignment, and similarly for the assignments $n_S := \dots$ for the different S 's. We will denote again with $r_c^\infty, \bar{r}_c^\infty$ and n_S^∞ the limit of the values of the corresponding variables r_c, \bar{r}_c and n_S in the execution process.

The corresponding correctness and completeness proofs for this case are not completely trivial extensions of the ones for the left-linear case, but follow the same scheme.

Theorem 3. (*correctness*) *For every c and every constrained term $(t|C)$ in r_c^∞ , there exists at least one instance of $(t|C)$, and all instances of $(t|C)$ are innermost-reachable from c .*

For every c and every constrained term $(t|C)$ in \bar{r}_c^∞ , all non-constant normal form instances of $(t|C)$ are innermost-reachable from c .

For every $S \in \mathcal{P}(\Sigma_0)$ and every term t in n_S^∞ , it holds that t is a non-constant normal form innermost-reachable from S .

Theorem 4. (*completeness*) *For every constant c and every weak normal form s reachable from c , there exists some constrained term $t|C$ in r_c^∞ such that s is an instance of $t|C$, and $\forall x \in \text{Vars}(t), d \in C(x) : |d \rightarrow^* x\sigma| < |c \rightarrow^* s|$.*

For every constant c and every non-constant normal form s innermost-reachable from c , there exists some constrained term $ft_1 \dots t_m|C$ in \bar{r}_c^∞ such that s is an instance of $ft_1 \dots t_m|C$.

For every $S \in \mathcal{P}(\Sigma_0)$ and every normal form t innermost-reachable from S , the term t belongs to n_S^∞ .

We will not use the following lemma in the rest of the article. We include it here as a consequence of the completeness Theorem and the WNF-representation process itself, since it is interesting because establishes that the WNF-representation process computes a representative for every weak normal form that is maximal in a certain sense.

Lemma 3. *If $ft_1 \dots t_m$ is a weak normal form innermost-reachable from a constant c , then there exists a constrained term $f\alpha_1 \dots \alpha_m|C$ in r_c^∞ satisfying the following for every t_i . If t_i is a constant then $\alpha_i = t_i$. If t_i is a non-constant normal form, then α_i is a variable and $C(\alpha_i)$ is the set of all constants that innermost-reach t_i .*

6.2 The Bounded WNF-Representation Process

The WNF-representation process gives a representation of all weak normal forms reachable from any constant. The problem is that it may not terminate. The sets

n_S can increase indefinitely, and their new elements can contribute to generate other new normal forms, and to change other sets $n_{S'}$ to be not empty. If at some point of the execution, no set n_S is going to change from empty to non-empty any more, then, the sets r_c and \bar{r}_c will not change any more during the rest of the execution. This is because when jumping from step 5 to step 2, the conditions of the sets in the assignment of step 2 have not changed. Hence, if we give a criteria for detecting that no n_S is going to change from empty to non-empty any more, we will be able to algorithmically determine the sets r_c and \bar{r}_c . To this end, we define the *bounded WNF-representation process* to be the same WNF-representation process, but bounding the number of elements to be contained in the variable sets n_S . We allow them to contain at most $m * |\mathcal{P}(\Sigma_0)|$ elements, i.e. $m * (2^{|\Sigma_0|} - 1)$. Hence the union \cup operator in the assignment of step 4. has to be changed to a modified union $\bar{\cup}$, that is interpreted as any operator such that $A \subseteq A\bar{\cup}B \subseteq A \cup B$, $|A\bar{\cup}B| \leq m * (2^{|\Sigma_0|} - 1)$, where $|A\bar{\cup}B|$ is maximal under these conditions.

The bounded WNF-representation process terminates, since the sets r_c , \bar{r}_c and n_S can not increase indefinitely, and when none of them changes in the corresponding steps 2. and 4., the process ends.

We will call r_c^{bnd} , \bar{r}_c^{bnd} and n_S^{bnd} to every set r_c , \bar{r}_c and n_S at the end of the bounded WNF-representation process. It is clear that every r_c^{bnd} , \bar{r}_c^{bnd} and n_S^{bnd} is included into its corresponding r_c^∞ , \bar{r}_c^∞ and n_S^∞ .

Suppose that we would execute a variation of the (non-terminating) WNF-representation process consisting on changing step 1. to the new instruction:

1. For every constant c do $r_c := r_c^{bnd}$ and $\bar{r}_c := \bar{r}_c^{bnd}$, and for every $S \in \mathcal{P}(\Sigma_0)$ do $n_S := n_S^{bnd}$.

In the limit we would obtain again r_c^∞ and n_S^∞ for every corresponding variable r_c , \bar{r}_c and n_S . We will repeatedly use this fact when proving the following lemma and corollary, referring to this variant as the *bound-starting WNF-representation process*.

Lemma 4. $|n_S^{bnd}| \geq \text{minimum}(|n_S^\infty|, m)$

Proof. For proving the statement, we will prove inductively on k that, if there are at least k sets S in $\mathcal{P}(\Sigma_0)$ such that $n_S^{bnd} \neq n_S^\infty$, then, there exists at least k sets S in $\mathcal{P}(\Sigma_0)$ such that $n_S^{bnd} \neq n_S^\infty$ and $|n_S^{bnd}| \geq m * |\mathcal{P}(\Sigma_0)| - m * (k - 1)$. This automatically implies that $|n_S^{bnd}| \geq \text{minimum}(|n_S^\infty|, m)$ holds for every $S \in \mathcal{P}(\Sigma_0)$, since $m * |\mathcal{P}(\Sigma_0)| - m * (k - 1) \geq m$ for all such k .

Hence, assume that there are at least k sets S in $\mathcal{P}(\Sigma_0)$ such that $n_S^{bnd} \neq n_S^\infty$. If $k = 1$ the result trivially holds, since $n_S^{bnd} \neq n_S^\infty$ for some set S implies that some set has reached $m * |\mathcal{P}(\Sigma_0)|$ elements during the bounded WNF-representation process. If $k > 1$, by induction hypothesis, there exists $k - 1$ sets S in $\mathcal{P}(\Sigma_0)$ such that $n_S^{bnd} \neq n_S^\infty$ and $|n_S^{bnd}| \geq m * |\mathcal{P}(\Sigma_0)| - m * (k - 2)$. It rests to show that there is an other set S' different from these $k - 1$ sets such that $n_{S'}^{bnd} \neq n_{S'}^\infty$ and $|n_{S'}^{bnd}| \geq m * |\mathcal{P}(\Sigma_0)| - m * (k - 1)$. We prove it by contradiction assuming that such S' does not exist. Hence, at this point we can classify the sets of $\mathcal{P}(\Sigma_0)$

into three kinds. The sets of kind (i) are those $k - 1$, i.e. the sets S such that $n_S^{bnd} \neq n_S^\infty$ and $|n_S^{bnd}| \geq m * |\mathcal{P}(\Sigma_0)| - m * (k - 2)$. The sets of kind (ii) are the sets S' such that $n_{S'}^{bnd} \neq n_{S'}^\infty$ and $|n_{S'}^{bnd}| < m * |\mathcal{P}(\Sigma_0)| - m * (k - 1)$. The sets of kind (iii) are the sets S'' such that $n_{S''}^{bnd} = n_{S''}^\infty$. If we execute the bound-starting WNF-representation process, at some execution point T there is a set S' of kind (ii) such that $n_{S'}$ increases for the first time, i.e. for every $\overline{S'}$ of kind (ii), $n_{\overline{S'}}$ preserves its initial value until T . The assignment that increases $n_{S'}$ is the one of item 4., that we write as follows indicating which values coincide with the ending values in the bounded WNF-representation process (for example, note that no \overline{r}_d nor r_d has changed until T , since no set n_S has changed from empty to non-empty until T ; hence, the chosen $t|C$ is also in all \overline{r}_d^{bnd}).

$$n_{S'} := n_{S'}^{bnd} \cup \{t\sigma \mid \exists C : \forall d \in S' : (t|C) \in \overline{r}_d^{bnd} \text{ up to renaming, and} \\ \forall x \in \text{Dom}(C) : (x\sigma \in n_{C(x)}) \text{ and} \\ t\sigma \text{ is a non-constant normal form } \}$$

Let $x_1 \dots x_k$ be the variables of $\text{Dom}(C)$ ordered in any way such that, if $C(x_i)$ is of kind (ii) or (iii) and $C(x_j)$ is of kind (i), then $i < j$. We construct a new substitution θ defined on these variables, recursively from x_1 to x_k . We consider the variable x_i and assume that θ is already defined for the previous ones. If $C(x_i)$ is of kind (ii) or (iii), then we define $x_i\theta := x_i\sigma$ (note that in this case $x_i\sigma \in n_{C(x_i)}^{bnd}$). If $C(x_i)$ is of kind (i), we chose p to be any of the positions such that $t|_p = x_i$, and we chose any term t_i of $n_{C(x_i)}^{bnd}$ different from all terms in $\{s|_p \mid s \in n_{S'}^{bnd}\}$ and different from all $x_1\theta, \dots, x_{i-1}\theta$ (note that this is possible since $m < |n_{C(x_i)}^{bnd}| - |n_{S'}^{bnd}|$). By the construction of θ , every $x_i\theta$ belongs to $n_{C(x_i)}^{bnd}$, and $t\theta$ is different from all terms in $n_{S'}^{bnd}$.

Note that all the x_i are replaced to non-constant normal forms by θ , as σ does, and in the variables where θ and σ differ, θ replaces each of these variables by a term different from all the rest of term replacements by θ . Hence, $t\theta$ is a non-constant normal form since $t\sigma$ is.

Previous facts imply that the term $t\theta$ will be added to $n_{S'}$ during the execution of the bounded WNF-representation process, and hence, that $t\theta$ belongs to $n_{S'}^{bnd}$. But they also imply that $t\theta$ does not belong to $n_{S'}^{bnd}$, and this is a contradiction. \square

Corollary 1. *For every $S \in \mathcal{P}(\Sigma_0)$, it holds that n_S^{bnd} is empty if and only if n_S^∞ is empty. Moreover, for every $c \in \Sigma_0$, it holds that $r_c^{bnd} = r_c^\infty$ and $\overline{r}_c^{bnd} = \overline{r}_c^\infty$.*

Proof. The first statement is a direct consequence of Lemma 4. The second statement is a direct consequence of the first, and the fact that the bound-starting WNF-representation process starts with $r_c = r_c^{bnd}$ and $\overline{r}_c = \overline{r}_c^{bnd}$, and will not produce any modification of the r_c 's and \overline{r}_c 's. This is because the assignment in step 2. of the WNF-representation process introduces new elements only when some n_S has changed from empty to non-empty. \square

7 Decidability of Reachability

Given two terms s and t and a flat TRS R , we want to check whether t is innermost-reachable from s by R . We assume that s and t are ground: otherwise the variables can be replaced by new constants preserving the answer.

First of all we show how this problem can be reduced to a simpler one, where one can assume that s is a constant. Given an input $\langle s, t, R \rangle$ of the general reachability problem, we construct a new constant $c_{s'}$ for every subterm s' of s . For every subterm of s of the form $f s_1 \dots s_m$ we construct a new rule $c_{f s_1 \dots s_m} \rightarrow f c_{s_1} \dots c_{s_m}$. For every constant subterm d of s we construct a new rule $c_d \rightarrow d$. Let R' be the TRS R extended with these new rules. Then, R' is also flat, and $s \rightarrow_R^* t$ if and only if $c_s \rightarrow_{R'}^* t$. The only if part is obvious, and the if part is a straightforward consequence of the fact that $u \rightarrow_{R'} v$ implies $\text{NF}_{R'-R}(u) \rightarrow_R^{0,1} \text{NF}_{R'-R}(v)$, where $\text{NF}_{R'-R}(_)$ refers to the unique normal form with respect to $R' - R$. Hence, the initial problem can be reduced to a simpler instance $\langle c, t, R \rangle$.

input: $\langle c, t, R \rangle$

1. compute all r_d 's with the "WNF representation algorithm" with R . Let T be the set of all subterms of t .
2. $\text{REACH} := \{ \langle d, e \rangle \mid d, e \in \Sigma_0 \wedge e \notin \emptyset \in r_d \}$.
3. $\text{REACH} := \text{REACH} \cup$

$$\{ \langle d, f s_1 \dots s_m \rangle \in \Sigma_0 \times T \mid \exists f s'_1 \dots s'_m \mid C \in r_d : \forall i, j \in \{1 \dots m\} : \\ (s'_i \notin \Sigma_0 \wedge s'_i = s'_j \Rightarrow s_i = s_j) \wedge \\ (s'_i \in \Sigma_0 \wedge \langle s'_i, s_i \rangle \in \text{REACH}) \vee \\ (s'_i \notin \Sigma_0 \wedge s_i \text{ is a normal form} \wedge \\ \forall e \in C(s'_i) : \langle e, s_i \rangle \in \text{REACH}) \} \cup$$

$$\{ \langle d, s \rangle \in \Sigma_0 \times T \mid \exists x \{ \langle x, S \rangle \} \in r_d : \forall e \in S : \\ ((e, s) \in \text{REACH} \wedge s \text{ is a normal form}) \}$$

4. If REACH has changed then repeat previous step.
5. Give positive answer if $\langle c, t \rangle \in \text{REACH}$, and negative answer otherwise.

Lemma 5. *At the end of the previous algorithm, REACH contains exactly the pairs $\langle d, s \rangle$ such that d is a constant, s is a subterm of t , and $d \rightarrow^* s$.*

8 Decidability of Joinability

Given two terms s and t and a flat TRS R , we want to check whether s and t are innermost-joinable by R . As before, we can assume that s and t are ground.

Similarly to the previous section, this problem can be reduced to joinability of constants. The transformation is identical: we construct a new constant c_u for every subterm u of s or t , and extend R to R' by adding a new rule $c_{f u_1 \dots u_m} \rightarrow f c_{u_1} \dots c_{u_m}$ for every subterm $f u_1 \dots u_m$ of s or t , and a new rule $c_d \rightarrow d$ for

every constant subterm d of s or t . The instance $\langle s, t, R \rangle$ is then transformed into $\langle c_s, c_t, R' \rangle$.

In the following we describe an algorithm to solve this simplified problem.

input: $\langle c_1, c_2, R \rangle$

1. compute all r_d 's and n_d^{bnd} 's with the "WNF representation algorithm" with R .
2. make $\text{JOIN} := \{ \langle c, d \rangle \mid \exists c' | \emptyset \in r_c \cap r_d \} \cup \{ \langle c, d \rangle \mid n_{\{c,d\}}^{bnd} \neq \emptyset \}$
3. $\text{JOIN} := \text{JOIN} \cup$

$$\{ \langle c, d \rangle \mid ; \exists f s_1 \dots s_m | C \in r_c, f t_1 \dots t_m | D \in r_d : \forall 1 \leq i \leq m : \\ ((s_i, t_i \in \Sigma_0 \wedge \langle s_i, t_i \rangle \in \text{JOIN}) \vee \\ (s_i \in \Sigma_0, t_i \notin \Sigma_0 \wedge n_{\{s_i\} \cup D(t_i)}^{bnd} \neq \emptyset) \vee \\ (s_i \notin \Sigma_0, t_i \in \Sigma_0 \wedge n_{C(s_i) \cup \{t_i\}}^{bnd} \neq \emptyset) \vee \\ (s_i, t_i \notin \Sigma_0 \wedge n_{C(s_i) \cup D(t_i)}^{bnd} \neq \emptyset)) \}$$
4. If JOIN has changed then repeat previous step.
5. Give positive answer if $\langle c_1, c_2 \rangle \in \text{JOIN}$, and negative answer otherwise.

Lemma 6. *The algorithm given above decides if two constants are innermost-joinable.*

The previous algorithm is not useful if we restrict again to the more efficient algorithm for the left-linear case, since the WNF-representation algorithm provided in Section 5 computes the sets r_c , but not the sets n_s . Nevertheless, note that for this case we just need to know if sets of the form $n_{\{c,d\}}$ are empty, i.e. if two constants c and d reach a common non-constant normal form. The following algorithm computes all pairs of constants satisfying this property.

input: R

1. compute all r_d 's with the "WNF representation algorithm for left-linear TRS" with R .
2. $\text{JOINNF} := \emptyset$
3. $\text{JOINNF} := \text{JOINNF} \cup$

$$\{ \langle c, d \rangle \mid \exists f s_1 \dots s_m | C \in r_c, f t_1 \dots t_m | D \in r_d : \\ (f s_1 \dots s_m \text{ is a normal form, and } \forall 1 \leq i \leq m : \\ ((s_i, t_i \in \Sigma_0 \wedge s_i = t_i) \vee \\ (s_i, t_i \notin \Sigma_0 \wedge \exists c_i, d_i \in \Sigma_0 : (C(s_i) = \{c_i\} \wedge D(t_i) = \{d_i\} \wedge \\ \langle c_i, d_i \rangle \in \text{JOINNF}))) \}$$

$$\{ \langle c, d \rangle \mid \exists x | \{ (x, \{c'\}) \} \in r_c : \langle c', d \rangle \in \text{JOINNF} \}$$

$$\{ \langle c, d \rangle \mid \exists x | \{ (x, \{d'\}) \} \in r_d : \langle c, d' \rangle \in \text{JOINNF} \}$$
4. If JOINNF has changed then repeat previous step.

Lemma 7. *At the end of the execution of the previous algorithm with a left-linear TRS, JOINNF contains all pairs $\langle c, d \rangle$ such that c and d innermost-reach a common non-constant normal form t .*

9 Complexity

Given a flat TRS R with size bounded by n , the bounded WNF-representation process requires to compute the sets r_c and n_S . There are at most n constants, and at most $|\mathcal{P}(\Sigma_0)|$ sets, i.e. $(2^{|\Sigma_0|} - 1)$. The elements in every set r_c are of the form $f\alpha_1 \dots \alpha_m|C$, where every α_i is either a constant, or a variable that coincides with some of the previous $\alpha_1 \dots \alpha_{i-1}$, or a different variable from previous ones and with a related set in C . Hence, the number of elements in every set r_c is bounded by $m * (n + m + |\mathcal{P}(\Sigma_0)|)$. By the conditions of the bounded WNF-representation algorithm, every n_S will contain at most $m * (2^{|\Sigma_0|} - 1)$ elements. The assignments do simple operations of matching on these data, and hence, their cost is $2^{\mathcal{O}(n)}$. Since the bounded WNF-representation process ends when no set is changed any more, the number of execution steps is also bounded by the maximum size of the sets r_c 's and n_S 's. Altogether implies that this algorithm is exponential ($2^{\mathcal{O}(n)}$) in size and space.

Our algorithms for reachability and joinability add first some new constants and rules, but less than the size of the input (n changes to $2n$ at most), and then, require the computation of the WNF representation. After that, the algorithm for reachability computes at most if every constant reaches every subterm of t . this is done recursively, and every of such checks needs again some simple operations of matching on the WNF representation. This is similar for joinability, where all pairs of constants are considered at every step of the algorithm and there are at most as steps as pairs of constants. Again, every step corresponds to some operations of matching on the WNF representation. Altogether implies that these algorithms are exponential ($2^{\mathcal{O}(n)}$) in size and space.

Exptime-hardness of these problems can be shown with an identical proof to the one given in [7] for the general (not necessarily innermost) case.

Theorem 5. *The innermost-reachability and innermost-joinability problems for shallow TRS are EXPTIME-complete, when the maximum arity m of the signature is fixed for the problem.*

With respect to the left-linear case, in Section 5 we have already seen that the time complexity of the WNF-representation algorithm is $(n+m)^{\mathcal{O}(m)}$, and hence, polynomial time if we assume that the maximum arity m of the signature is a constant. The algorithms of Sections 7 and 8 for computing the sets REACH, JOIN and JOINNF are just fixpoint computations of pairs of elements from sets with polynomially bounded size on the input. The operations at every step of the fixpoint computation consist in searches on the sets REACH, JOIN and JOINNF, and on the WNF-representation. Hence, we conclude that the time complexity is again polynomial for the reachability and joinability problems.

Theorem 6. *The innermost-reachability and innermost-joinability problems for shallow left-linear TRS are decidable with polynomial time complexity, when the maximum arity m of the signature is fixed for the problem.*

References

1. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (1997) Available on <http://www.grappa.univ-lille3.fr/tata>
2. Coquidé, J.L., Dauchet, M., Gilleron, R., Vágvölgyi, S.: Bottom-up tree pushdown automata: classification and connection with rewrite systems. *Theoretical Computer Science* 127, 69–98 (1994)
3. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: van Leeuwen, J.(ed.) *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, pp. 243–320, Amsterdam, North-Holland (1990)
4. Ganzinger, H., Jacquemard, F., Veanes, M.: Rigid reachability: The non-symmetric form of rigid E-unification. *Intl. Journal of Foundations of Computer Science* 11(1), 3–27 (2000)
5. Godoy, G., Tiwari, A.: Deciding fundamental properties of right-(ground or variable) rewrite systems by rewrite closure. In: Basin, D., Rusinowitch, M. (eds.) *IJCAR 2004. LNCS (LNAI)*, vol. 3097, pp. 91–106. Springer, Heidelberg (2004)
6. Godoy, G., Tiwari, A.: Termination of rewrite systems with shallow right-linear, collapsing, and right-ground rules. In: Nieuwenhuis, R. (ed.) *Automated Deduction – CADE-20. LNCS (LNAI)*, vol. 3632, pp. 164–176. Springer, Heidelberg (2005)
7. Godoy, G., Tiwari, A., Verma, R.: On the confluence of linear shallow term rewrite systems. In: Alt, H., Habib, M. (eds.) *STACS 2003. LNCS*, vol. 2607, pp. 85–96. Springer, Heidelberg (2003)
8. Gyenizse, P., Vgvlgyi, S.: Linear generalized semi-monadic rewrite systems effectively preserve recognizability. *Theoretical Computer Science* 194, 87–122 (1998)
9. Jacquemard, F.: Reachability and confluence are undecidable for flat term rewriting systems. *Inf. Process. Lett.* 87(5), 265–270 (2003)
10. Nagaya, T., Toyama, Y.: Decidability for left-linear growing term rewriting systems. In: Narendran, P., Rusinowitch, M. (eds.) *RTA 1999. LNCS*, vol. 1631, pp. 256–270. Springer, Heidelberg (1999)
11. Salomaa, K.: Deterministic tree pushdown automata and monadic tree rewriting systems. *J. Comput. Syst. Sci.* 37, 367–394 (1998)
12. Takai, T., Kaji, Y., Seki, H.: Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In: Bachmair, L. (ed.) *RTA 2000. LNCS*, vol. 1833, pp. 246–260. Springer, Heidelberg (2000)
13. Takai, T., Seki, H., Fujinaka, Y., Kaji, Y.: Layered transducing term rewriting system and its recognizability preserving property. *IEICE Transactions on Information and Systems* E86-D(2), 285–295 (2003)

Termination of Rewriting with Right-Flat Rules^{*}

Guillem Godoy¹, Eduard Huntingford², and Ashish Tiwari³

¹ Technical University of Catalonia
Jordi Girona 1, Barcelona, Spain
ggodoy@lsi.upc.edu

² Technical University of Catalonia
Jordi Girona 1, Barcelona, Spain
eduard.hl@gmail.com

³ SRI International, Menlo Park, CA 94025
tiwari@csl.sri.com

Abstract. Termination and innermost termination are shown to be decidable for term rewrite systems whose right-hand side terms are restricted to be shallow (variables occur at depth at most one) and linear. Innermost termination is also shown to be decidable for shallow rewrite systems. In all cases, we show that nontermination implies nontermination starting from flat terms. The proof is completed by using the useful enabling result that, for right shallow rewrite systems, existence of nonterminating derivations starting from a given term is decidable. We also show that termination is undecidable for shallow rewrite systems. For right-shallow systems, general and innermost termination are both undecidable.

1 Introduction

Termination is an important property of computing systems and it has generated significant renewed interest in recent years. There has been progress on both the theoretical and practical aspects of proving termination of many different computing paradigms - such as term rewrite systems, functional programs, and imperative programs. Innermost termination refers to termination of rewriting restricted to the innermost strategy, which forces the complete evaluation of all the subterms before the rule application at a position. It corresponds to the "call by value" computation of programming languages. A typical example of a rewrite system that is innermost terminating but not terminating is the following [12].

$$\{f(0, 1, x) \rightarrow f(x, x, x), c \rightarrow 0, c \rightarrow 1\}$$

^{*} The first two authors were supported by Spanish Min. of Educ. and Science by the LogicTools project (TIN2004-03382). The second author was also supported by Spanish Min. of Educ. and Science by the GRAMMARS project (TIN2004-07925-C03-01). The third author was supported in part by the National Science Foundation under grant CCR-0326540.

The non-terminating derivation

$$\underline{f(0, 1, c)} \rightarrow f(\underline{c}, \underline{c}, c) \rightarrow^2 f(0, 1, c) \dots$$

is not possible with innermost rewriting, since c has to be normalized before any rule application at position λ in $f(0, 1, c)$.

This paper focuses on termination and innermost termination of term rewrite systems (TRS). For results on termination of imperative programs, the reader is referred to [6,11,2,11]. There has been steady progress on the problem of deciding if a term rewrite system is terminating. While termination is undecidable for general term rewrite systems and string rewrite systems [7], several subclasses with decidable termination problem have been identified. Termination is decidable in polynomial time for ground term rewrite systems [7,9]. Termination is decidable for right-ground term rewrite systems [3] and also for the more general class that also has right-variable rules [4]. Later, termination was shown to be decidable for rewrite systems that contain right-ground, collapsing, or shallow right-linear rewrite rules [5]. There were further decidability results about shallow left-linear and shallow right-linear rewrite systems [13].

This paper presents a new decidability result that subsumes some of the above results. In particular, we show that termination (and innermost termination as well) is decidable for rewrite systems whose right-hand side terms are both shallow and linear, that is, variables occur at depth at most one and no variable occurs more than once on the right-hand side terms. There is no restriction on the left-hand side terms. Thus, right-ground systems and shallow right-linear systems are both contained in our class.

The proofs presented in this paper are simple, self-contained and not dependent on any other results. At the top-level, we argue that if a right flat-linear rewrite system is nonterminating, then there is a nonterminating derivation starting from a constant or a ground flat term (Section 6). To complete the proof, we prove a (stronger) result: termination of right shallow systems *starting from a given term* is decidable (Section 4). In other words, nontermination is semi-decidable for right shallow systems. Using similar arguments, we also show that innermost termination of flat term rewrite systems is decidable (Section 5). This last result is in sharp contrast to the undecidability of general termination for flat rewrite systems (Section 7). We also show undecidability of general and innermost termination for right shallow rewrite systems (Section 7).

2 Preliminaries

A *signature* $\Sigma = \cup_i \Sigma_i$ is a finite set of function symbols and constants indexed by their *arity* i . Thus, Σ_0 is the set of constants in Σ . A *term* t over Σ is constructed from the symbols in Σ and a set of variables in the usual way. To reduce clutter, we write $ft_1 \dots t_m$ instead of the standard notation $f(t_1, \dots, t_m)$ for a term. A *position* is a string of natural numbers (including the empty string λ). We use $.$ as the string concatenation operator and $|p|$ to denote the length of the string p . Given a term t and a position p , the subterm of t at position p ,

denoted by $t|_p$, is defined to be (i) t , if $p = \lambda$, (ii) $t_i|_{p'}$ if $t = ft_1, \dots, t_m$ and $p = i.p'$ and $1 \leq i \leq m$, and (iii) \perp , otherwise. The set $\mathcal{Pos}(t)$ consists of all positions p such that $t|_p \neq \perp$. The subterm $t|_p$ of t is said to be *at depth* $|p|$.

We write $p_1 > p_2$ (equivalently, $p_2 < p_1$) and say p_1 is below p_2 (equivalently, p_2 is above p_1) if p_2 is a proper prefix of p_1 , that is, $p_1 = p_2.p'_2$ for some nonempty p'_2 . Positions p and q are *disjoint* if $p \not\preceq q$ and $q \not\preceq p$. When p_1 is of the form $p_2.p'_2$, $p_1 - p_2$ denotes p'_2 .

The *height* of a term t , $height(t)$, is zero if t is a variable or a constant, and it is $1 + \max\{height(t_i) : i = 1, \dots, m\}$ if $t = ft_1 \dots t_m$. A term t is said to be *flat* if its height is at most one. A term t is *shallow* if all variables in t occur at depth at most one. A term t is *linear* if no variable occurs more than once in t .

The set of all subterms of a term s is denoted by $\lfloor s \rfloor$. A term t is said to be *reachable* from s if $s \rightarrow_R^* t$. A term t is said to be *context-reachable* from s if there exists a context $C[_]$ such that $s \rightarrow_R^* C[t]$.

A *rewrite system* R is a finite set of rewrite rules $l \rightarrow r$, where l and r are terms. A *rewrite step*, using the rule $l \rightarrow r$, applied to a term s at position p , is denoted by $s \rightarrow_{l \rightarrow r, \sigma, p} s[r\sigma]_p$, where σ is a substitution such that $l\sigma \equiv s|_p$. We do the usual assumptions for the rules $l \rightarrow r$ of a rewrite system R , i.e. l is not a variable, and all variables occurring in r also occur in l . A rewrite system R is *terminating from* s if there are no infinite R -derivations, $s \rightarrow_R s_1 \rightarrow_R \dots$. If R is terminating from every term, then R is said to be *terminating*.

A term s is *R -irreducible* if there is no term t such that $s \rightarrow_R t$. A rewrite step $s \rightarrow_{R,p} t$ is an *innermost rewrite step* if $s|_{p'}$ is R -irreducible, for all $p' > p$. The concepts of reachability and termination can be naturally defined for innermost rewriting.

A rewrite system R is (right-)shallow, respectively flat/linear, if all (right-hand side) terms in R are shallow, respectively flat/linear.

3 Flattening and Other Simplifying Assumptions

The discussion of this section is written for general termination, but it is also valid when we interpret termination as innermost termination, reachability as innermost reachability, and so on. To this end, in the innermost case we assume that for a given TRS, all the rules $l \rightarrow r$ such that l has a proper subterm that is not a normal form have been removed. Note that these rules can not be used in an innermost derivation.

For purposes of deciding termination of R , we show that we can assume, without loss of any generality, that all shallow terms in the rewrite system R are indeed flat. This observation follows from the following result.

Proposition 1. *Let R be a rewrite system containing a rewrite rule $l[u] \rightarrow r[v]$ such that u and v are ground. Let c and d be two new constants. Then, R is (innermost-)terminating iff $R' := R - \{l \rightarrow r\} \cup \{l[c] \rightarrow r[d], u \rightarrow c, d \rightarrow v\}$ is (innermost-)terminating.*

Proof. (Sketch) Any rewrite step $s \rightarrow_R t$ can be simulated by a R' -derivation: if the R rewrite step does not use $l[u] \rightarrow r[v]$, then $s \rightarrow_{R'} t$; if it does, then

$s := s[l[u]]$ and $t := s[r[v]]$ and the corresponding R' -derivation is $s[l[u]] \rightarrow s[l[c]] \rightarrow s[r[d]] \rightarrow s[r[v]]$.

Any R' -derivation $s_1 \rightarrow_{R'} s_2 \rightarrow_{R'} \dots$ can be simulated by the R -derivation $s_1 \sigma \rightarrow_R^* s_2 \sigma \rightarrow_R^* \dots$, where σ is the substitution $\{c \mapsto u, d \mapsto v\}$. It can be seen that infinite R' -derivations necessarily map to infinite R -derivations. \square

As a second simplifying assumption, we note that we can assume, again without loss of generality, that the signature Σ of R contains constants and exactly one m -ary function symbol. If Σ contains multiple n -ary function symbols ($n > 0$), then we choose m to be one plus the maximum arity of a function symbol in Σ . Let $\Sigma' := \Sigma'_0 \cup \Sigma'_m$, where $\Sigma'_0 := \Sigma$ and $\Sigma'_m = \{f\}$. Now, if $g(s_1, \dots, s_n)$ is a term over Σ , we encode it by the term $f(\text{encode}(s_1), \dots, \text{encode}(s_n), g, \dots, g)$ over Σ' , where $\text{encode}(s_i)$ recursively encodes s_i and the $g \in \Sigma'_0$ is used as padding to get m arguments for f . This encoding of terms can be extended to rewrite systems. It is easy to see that this encoding preserves the termination property of rewrite systems.

4 Right Flat Systems

In this section, we will show that, given a right shallow rewrite system R and a term s , it is decidable if R is terminating from s . In particular, this implies that nontermination is semi-decidable for right shallow rewrite systems. We will show that termination is undecidable for right shallow systems in Section 7. A right shallow system can be transformed into a right flat rewrite system, while preserving its termination property, similarly to what is shown in Section 3. Henceforth, in this section, R is a right flat TRS.

The proofs of this section are written for general termination, but they are also valid when we interpret termination as innermost termination, reachability as innermost reachability, and so on.

An important property of a right flat system R is that if $s \rightarrow_R^* t$, then every subterm of t is reachable from either a constant or some subterm of s . We prove this and other useful consequences by inductively marking each position of a term (in the above derivation) by the witness term from $[s] \cup \Sigma_0$ as follows:

Base Case: For the term s , the marking is given by $M_s(p) = s|_p$ for all positions p of s .

Induction Step: Consider the rewrite step, $u \rightarrow_{l \rightarrow r, p} v$, where we have the mapping $M_u : \text{Pos}(u) \mapsto [s] \cup \Sigma_0$. We construct the mapping $M_v : \text{Pos}(v) \mapsto [s] \cup \Sigma_0$ as follows:

1. if $p' \not\prec p$, then $M_v(p') := M_u(p')$,
2. if $p' = p.p_0$, $|p_0| = 1$, and $r|_{p_0}$ is a constant, then $M_v(p') := r|_{p_0}$.
3. for every p_0 such that $r|_{p_0}$ is a variable we choose any position q_0 in l such that $l|_{q_0} = r|_{p_0}$ and then for every $p' = p.p_0.p_1$, $|p_0.p_1| \geq 1$, we define $M_v(p') := M_u(p.q_0.p_1)$.

Note that we are assuming that every variable on the right-hand side also appears on the left-hand side; if not, then the rewrite system is trivially nonterminating. One easy property of the markings is that $v|_p$ is reachable from $M_v(p)$.

Lemma 1. *For a right flat R , if the derivation $s \rightarrow_R^* t$ is marked as above, then $t|_p$ is (innermost-)reachable from $M_t(p)$ for all $p \in \mathcal{Pos}(t)$. In particular, if s is a constant, all subterms of t are (innermost-)reachable from a constant.*

Moreover, $M_t(\lambda) = s$, and if s is not a constant, then for any $p \in \mathcal{Pos}(t)$ with $p \neq \lambda$ we have $M_t(p) \neq s$.

Proof. The claim is proved inductively on the length of the derivation $s \rightarrow_R^* t$. The base case is trivial. For the induction step, suppose $u \rightarrow_{l \rightarrow r, p} v$ and $u|_{p'}$ is reachable from $M_u(p')$ for all $p' \in \mathcal{Pos}(u)$. Consider any $p' \in \mathcal{Pos}(v)$. We show that $v|_{p'}$ is reachable from $M_v(p')$ as follows:

- If $p' \leq p$, then we have $M_v(p') = M_u(p') \rightarrow_R^* u|_{p'} \rightarrow_{l \rightarrow r, p-p'} v|_{p'}$.
- If $p' = p.p_0$, $|p_0| = 1$, and $r|_{p_0}$ is a constant, then $M_v(p') = r|_{p_0}$ is reachable from $v|_{p'} = r|_{p_0}$.
- If $p' = p.p_0.p_1$, $|p_0.p_1| \geq 1$, and $r|_{p_0}$ is a variable, then, for some q_0 , $M_v(p') = M_u(p.q_0.p_1)$, $v|_{p'} = u|_{p.q_0.p_1}$, and $u|_{p.q_0.p_1}$ is reachable from $M_u(p.q_0.p_1)$ by induction hypothesis.
- If $p' \parallel p$, then the claim holds by induction hypothesis again as $v|_{p'} = u|_{p'}$ and $M_v(p') = M_u(p')$.

The second statement of the lemma follows from the construction of the mapping function. This completes the proof. \square

A second property of markings is that $M_v(p)$ is context-reachable from $M_v(p')$ for all $p' \leq p$.

Lemma 2. *For a right flat R , if the (innermost-)derivation $s \rightarrow_R^* t$ is marked as above, then for all $p, p' \in \mathcal{Pos}(t)$ such that $p' < p$, $M_t(p)$ is (innermost-)context reachable from $M_t(p')$. Moreover, if $M_t(p)$ and $M_t(p')$ are both constants, then $M_t(p)$ is (innermost-)context reachable from $M_t(p')$ in one or more steps.*

Proof. The claim is proved inductively on the length of the derivation $s \rightarrow_R^* t$. The base case is trivial. For the induction step, suppose $u \rightarrow_{l \rightarrow r, p} v$ is the last step of the derivation and the claim is true for u . Consider any $p' \in \mathcal{Pos}(v)$. We prove the claim for all $p'' < p'$ as follows:

- If $p' \leq p$ or $p' \parallel p$, then we have $M_v(p') = M_u(p')$ and $M_v(p'') = M_u(p'')$ for all $p'' \leq p'$, and hence, by induction hypothesis, the claims follow.
- If $p' = p.p_0$, $|p_0| = 1$, and $r|_{p_0}$ is a constant, then $M_v(p') = r|_{p_0}$. Now consider any $p'' < p'$. By Lemma 1, $v|_{p''}$ is reachable from $M_v(p'')$. Note that $M_v(p')$ is a subterm of $v|_{p''}$. Hence, $M_v(p')$ is context reachable from $M_v(p'')$.
If $M_v(p'')$ is a constant, then $v|_{p''}$ is reachable from $M_v(p'')$ in one or more steps. Hence, $M_v(p')$ is context reachable from $M_v(p'')$ in one or more steps.
- If $p' = p.p_0.p_1$, $|p_0.p_1| \geq 1$ and $r|_{p_0}$ is a variable, then, for some q_0 , $M_v(p') = M_u(p.q_0.p_1)$. There are two cases. (a) If $p'' \leq p$, then $M_v(p'') = M_u(p'')$ and, by induction hypothesis, $M_u(p.q_0.p_1)$ is context reachable from $M_u(p'')$, which is the same as saying that $M_v(p')$ is context reachable from $M_v(p'')$.

(b) If $p'' > p$, then $M_v(p'') = M_u(p.q_0.p'_1)$ for some $p'_1 < p_1$ and, by induction hypothesis, $M_u(p.q_0.p_1)$ is context reachable from $M_u(p.q_0.p'_1)$. This is the same as saying that $M_v(p')$ is context reachable from $M_v(p'')$. In both cases, the second part of the claim is immediate by induction.

This completes the proof. \square

Finally, a third observation about the markings is that positions that are below $\text{height}(s)$ are always marked by constants.

Lemma 3. *For a right flat R , if the (innermost-)derivation $s \rightarrow_R^* t$ is marked as above, then for all $p \in \mathcal{P}os(t)$ such that $|p| > \text{height}(s)$, $M_t(p)$ is always a constant.*

Proof. This is proved inductively on the length of the derivation. For the base case, $t = s$ and the claim is vacuously true. We just need to show that the property is preserved by every rewrite step, say $u \rightarrow_{l \rightarrow r, p} v$. Consider any position $p' \in \mathcal{P}os(v)$ such that $|p'| > \text{height}(s)$.

(i) If $p' \parallel p$ or $p' \leq p$, then $M_v(p') = M_u(p')$ and by induction hypothesis $M_u(p')$ will be a constant.

(ii) If $p' = p.p_0$, $|p_0| = 1$ and $r|_{p_0}$ is a constant, then $M_v(p') = r|_{p_0}$, which is a constant.

(iii) If $p' = p.p_0.p_1$, $|p_0.p_1| \geq 1$ and $r|_{p_0}$ is a variable, then $M_v(p') = M_u(p.q_0.p_1)$ for some q_0 and $|p.q_0.p_1| \geq |p'|$. Hence induction hypothesis is applicable and we can conclude that $M_u(p.q_0.p_1)$, and therefore $M_v(p')$, is a constant. \square

A simple consequence of Lemma 2 and Lemma 3 is that if R is terminating from s , then the height of terms reachable from s is computationally bounded.

Corollary 1. *Let R be a right flat TRS, s any term. Then, if R is (innermost) terminating from s , then for any term t (innermost) reachable from s , we have $\text{height}(t) \leq \text{height}(s) + |\Sigma_0|$.*

Proof. We mark the derivation $s \rightarrow_R^* t$ as above. Now, suppose $\text{height}(t) > \text{height}(s) + |\Sigma_0|$. By Lemma 3, each position in t that is deeper than $\text{height}(s)$ is marked with a constant. Since $\text{height}(t) > \text{height}(s) + |\Sigma_0|$, by pigeon-hole principle, there are two positions $p, p' \in \mathcal{P}os(t)$ such that $p < p'$ and $M_t(p) = M_t(p')$ and $M_t(p)$ is a constant, say c . By Lemma 2, it follows that c is context reachable from c in one or more steps. Moreover, by Lemma 1 the position λ of every term in a derivation is marked with s . Using Lemma 2 again, we infer that $M_t(p)$, or c , is also context reachable from s . Thus, we have a derivation $s \rightarrow_R^* C_1[c] \rightarrow_R^+ C_1[C_2[c]] \rightarrow_R^+ C_1[C_2[C_2[c]]] \rightarrow_R^+ \dots$. Hence, there is an infinite derivation starting from s . \square

Using the above corollary, we can show that the existence of nonterminating derivations starting from a term is decidable for right flat systems.

Theorem 1. *Termination (innermost-termination) of a right flat TRS R from a given term is decidable. Hence, nontermination (innermost-nontermination) is semi-decidable for right flat rewrite systems.*

Proof. Let s be any term. We enumerate all derivations starting from s . If we reach a term with height greater than $\text{height}(s) + |\Sigma_0|$, then by corollary [□](#) we know that R is nonterminating from s . Otherwise, we will get only *finitely* many reachable terms. If there is a derivation that cycles among these terms, then R is nonterminating from s . If not, then R is terminating from s . \square

Remark: We state below a related and new (up to our knowledge) theorem that uses a similar argument as the proof of Theorem [□](#).

Theorem 2. *Let \mathcal{C} be a class of TRS's that are effectively regularity preserving, and s a term. Termination of the TRS's of \mathcal{C} from s is decidable.*

Proof. (sketch) The regular set S of terms reachable from s can be checked to be infinite, in which case we know that there exists an infinite derivation from s . Otherwise, we have a finite number of terms reachable from s , and we can check for the existence of a cycle. \square

We shall not use Theorem [□](#) in this paper. However, we note here that, using recent results on regularity preserving systems [\[10\]](#), we immediately get very simple proofs of known decidability results, such as for right-ground systems [\[3\]](#): a right-ground system is regularity preserving, and is non-terminating iff it is non-terminating from some right-hand side, which can be checked for every one using Theorem [□](#).

5 Innermost Termination of Flat TRS's

In this section, we show that innermost termination of flat rewrite systems is decidable. In sharp contrast, general termination is undecidable for flat rewrite systems (Section [7](#)).

Let R be a flat rewrite system. We show decidability of innermost termination of flat systems by showing that for nonterminating R , there is an infinite derivation starting with either a constant or a flat term. Using Theorem [□](#), we know that these checks are decidable.

Lemma 4. *Let R be a flat rewrite system that is not innermost terminating. Then, there is an infinite innermost derivation starting from a constant or a ground flat term.*

Proof. We assume that there is no infinite derivation from a constant and we show that there is one from a flat term.

Since R is not innermost terminating, there exists an infinite innermost derivation $t_0 \rightarrow_R t_1 \rightarrow_R \dots$ whose first step is at position λ . We first prove that for every i , every subterm at depth 1 of t_i is either reachable from a constant, or in normal form. First note that no term t_i is a constant, by our initial assumption. Now, we finish the proof of the claim by marking the derivation as above. By construction of marking, for any i , the mark at any depth 1 subterm of t_i is either a constant or a *proper* subterm of t_0 . Now, since we use innermost rewriting, all proper subterms of t_0 are in normal form. By Lemma [□](#), all subterms at

depth 1 of t_i are reachable from their markings, hence they are reachable from constants, or they are in normal form.

Now, we note that there exists at least one constant, call it c , that is in normal form. If not, any ground term can be innermost rewritten to another ground term, and hence there will be infinite derivations from constants, which contradicts our initial assumption.

We construct a new derivation $t'_0 \rightarrow_{R,\lambda} t'_1 \rightarrow_R \dots$ by defining each t'_i to be as t_i but replacing every subterm at depth 1 that is not reachable from any constant by the constant c chosen above. We need to show that the new derivation is “correct”, that is, there is a rewrite step from t'_{i-1} to t'_i . Consider the corresponding step $t_{i-1} \rightarrow_{l \rightarrow r, p} t_i$.

(i) If $p \neq \lambda$, then p occurs inside a subterm at depth 1 that is necessarily reachable from a constant, since otherwise this subterm would be in normal form. Therefore, the same rewrite step can be applied on t'_{i-1} to produce t'_i .

(ii) If $p = \lambda$, then, by our initial assumption, both l and r are not constants. Moreover, r can not be a variable, since, otherwise, t_i would be in normal form (and the derivation would be finite). Hence, $l \rightarrow r$ is of the form $f\alpha_1 \dots \alpha_m \rightarrow f\beta_1 \dots \beta_m$. If σ is the substitution used in this rewrite step, let σ' be as σ except for the cases where $x\sigma$ is not reachable from a constant, in which case we define $x\sigma' = c$. Clearly, $t'_{i-1} \rightarrow_{l \rightarrow r, \sigma', \lambda} t'_i$.

The infinite derivation $t'_0 \rightarrow t'_1 \rightarrow \dots$ is again innermost, and its initial term t'_0 satisfies that all their subterms at depth 1 are reachable from constants. Therefore, there exists a flat term s such that $s \rightarrow_R^* t'_0$, and hence, there exists an infinite derivation from a flat term s . \square

Lemma [4](#) and Theorem [1](#) together imply the following result.

Theorem 3. *Innermost termination is decidable for flat TRS's.*

Proof. Since there are only finitely many constants and ground flat terms, using Theorem [1](#), we can check if a given flat TRS R is not innermost terminating starting from one of these terms. By Lemma [4](#), we will find a witness for non-termination this way iff R is not innermost terminating.

6 Termination and Innermost Termination of Right Flat-Linear TRS's

A *right flat-linear* rewrite system has terms that are both flat and linear as right-hand side terms in all rewrite rules. In this section, we show decidability of termination and innermost termination for right flat-linear systems. Again, the proofs of this section are written for general rewriting, but they remain valid for innermost rewriting.

The proof of decidability of (innermost) termination for right flat-linear systems depends on two key observations. The first one is Lemma [1](#), which says that for any (innermost) derivation $s \rightarrow_R^* t$ using a right flat R , every *proper* subterm of t is (innermost) reachable from either a constant or a *proper* subterm of s .

The second key lemma is stated by first defining the following *measure* of a term t :

$$|t| := |\{p \mid t|_p \text{ is not (innermost) reachable from a constant}\}|$$

Note that $|t|$ depends on whether we are dealing with general or innermost rewriting. The next lemma uses right linearity of R as well.

Lemma 5. *Let R be a right flat-linear TRS. If s (innermost) rewrites to t , then $|s| \geq |t|$. Moreover, if $s[f s_1 \dots s_m]_p$ (innermost) rewrites to t at position p with a rule $fl_1 \dots l_m \rightarrow r$, and $|s| = |t|$, then, for every i in $\{1 \dots m\}$, if s_i is not (innermost) reachable from a constant, then l_i is a variable.*

Proof. Let $s \rightarrow_{l \rightarrow r, p} t$ be the rewrite step of the lemma. We prove the first statement by constructing an injective map, from positions p' of t such that $t|_{p'}$ is not reachable from a constant, to positions p'' of s such that $s|_{p''}$ is not reachable from a constant, as follows. If $p' \parallel p$ or $p' \leq p$, then we make $p'' := p'$. If $p' > p$, then p' can be written of the form $p.p_0.p_1$ where $r|_{p_0}$ is a height 0 term. In fact, $r|_{p_0}$ can not be a constant since otherwise $t|_{p'}$ would be a constant, and hence, it is a variable. We choose a position p'_0 such that $l|_{p'_0}$ is the same variable and set $p'' := p.p'_0.p_1$. The injectivity of the map follows by right linearity of R . Hence, $|s| \geq |t|$.

For the second statement, we are given that $|s| = |t|$, s is of the form $s[f s_1 \dots s_m]_p$ and l is of the form $fl_1 \dots l_m$. If a certain s_i is not reachable from a constant, but l_i is not a variable, then $p.i$ is not in the image of the previous mapping, and hence $|s| > |t|$, contradicting $|s| = |t|$. Therefore, all such l_i 's are variables. □

The idea of the decidability proof is the same as that for Theorem 3, that is, we show that if R is non-terminating, then it is non-terminating from a constant or a flat term.

Lemma 6. *Let R be a right flat and right linear rewrite system. If R is non-terminating (innermost-nonterminating), then there exists an infinite (innermost-) derivation from a constant or a flat ground term.*

Proof. Assume that there is no infinite derivation from a constant. We will show that there is one from a flat term.

Since R is non-terminating, there exists an infinite derivation $t_0 \rightarrow_R t_1 \rightarrow_R \dots$ that we choose to be the one with minimal height for t_0 . Note that no term t_i is a constant, by our initial assumption, and hence, if a rule of the form $l \rightarrow r$ is applied at λ , then neither l nor r is a constant.

First we show that *no collapsing rule is applied at λ* . This is immediate from Lemma 4: if $t_{i-1} \rightarrow_R t_i$ is the first collapsing rule applied at λ , then by Lemma 4, all subterms at depth 1 of t_{i-1} are either reachable from a constant or a proper subterm of t_0 . Since t_i is a proper subterm of t_{i-1} , it is reachable from either a

constant or a proper subterm of t_0 . In either case we have an infinite derivation starting from a term with smaller height than t_0 , which contradicts our choice of t_0 .

Now, we show that *there are infinite rewrite steps at position λ* . Suppose not. Let $t_{i-1} \rightarrow_R t_i$ be the last rewrite step at position λ . Then, there is an infinite derivation starting from some subterm at depth 1 of t_i . As before, this subterm is reachable from either a constant or a proper subterm of t_0 . Again, we have an infinite derivation that contradicts the minimality of t_0 .

Hence, the infinite derivation $t_0 \rightarrow_R \dots$ has infinite rewrite steps at position λ , and all those steps are using rules of the form $l \rightarrow f\alpha_1 \dots \alpha_m$, where the height of l is greater than or equal to 1. By Lemma 5, $|t_{i-1}| \geq |t_i|$ for all i . Since this relation can not be indefinitely decreasing, for some n we have $|t_n| = |t_{n+1}| = |t_{n+2}| = \dots$. From the infinite derivation $t_n \rightarrow_R t_{n+1} \rightarrow_R \dots$ we construct a new infinite derivation $t'_n \xrightarrow{0,1}_R t'_{n+1} \xrightarrow{0,1}_R \dots$ as follows. Analogously to the proof of Lemma 4, we can deduce that there exists at least one constant c that is a normal form (this is true for the innermost and the general case). For every t_i , we construct t'_i to be equal to t_i except for the subterms at depth 1 that are not reachable from constants, which are replaced by c . Formally, $t'_i = t_i[c]_{j_1} \dots [c]_{j_k}$ if $t_i|_{j_1}, \dots, t_i|_{j_k}$ are the subterms at depth 1 in t_i that are not reachable from constants.

We show that the new derivation is correct by analyzing each rewrite step $t_{i-1} \rightarrow_R t_i$ and its corresponding $t'_{i-1} \xrightarrow{0,1}_R t'_i$.

- (i) If $t_{i-1} \rightarrow_R t_i$ is at a position inside a subterm at depth 1 of t_{i-1} that is reachable from a constant, then, the same rewrite step can be applied on t'_{i-1} to produce t'_i .
- (ii) If $t_{i-1} \rightarrow_R t_i$ is at a position inside a subterm, say $t_{i-1}|_j$, at depth 1 of t_{i-1} that is not reachable from a constant, then, $t'_{i-1} = t'_i$. This is because $t_i|_j$ is also not reachable from a constant, as $|t_i|_j| = |t_{i-1}|_j| \geq 1$, and by Lemma 1, if $t_{i-1}|_j$ was reachable from a constant then all its subterms would be, and $|t_{i-1}|_j| = 0$. Hence, $t'_{i-1} \xrightarrow{0}_R t'_i$.
- (iii) If $t_{i-1} \rightarrow_R t_i$ is at position λ , then, by Lemma 5, if $fl_1 \dots l_m \rightarrow r$ and σ are the rule and substitution applied, then l_k is a variable for every position k such that $t_{i-1}|_k$ is not reachable from a constant. We define a new substitution σ' to be equal to σ except for such variables l_k , for which we define $l_k\sigma' = c$. The same rule $fl_1 \dots l_m \rightarrow r$ applied to t'_{i-1} at position λ and with substitution σ' produces t'_i .

Since every rewrite step $t_{i-1} \rightarrow_R t_i$ at position λ corresponds to a rewrite step $t'_{i-1} \xrightarrow{0,1}_R t'_i$, and there are infinite such steps, it follows that the derivation $t'_n \xrightarrow{0,1}_R t'_{n+1} \xrightarrow{0,1}_R \dots$ is infinite. Moreover, all subterms at depth 1 in t'_n are reachable from constants. Therefore, there exists a flat term t such that $t \xrightarrow{*}_R t'_n$, and hence, there exists an infinite derivation from a flat term t . □

Now, the main result follows immediately from Lemma 6 and Theorem 1.

Theorem 4. *Termination and innermost termination are both decidable for right shallow-linear rewrite systems.*

7 Undecidability Results

In this section, we prove undecidability of termination for shallow systems. This is done by a reduction from the Post correspondence problem (PCP) restricted to nonempty strings, i.e.:

$$\{\langle u_1, v_1 \rangle \dots \langle u_n, v_n \rangle \mid \forall 1 \leq i \leq n : (u_i \neq \lambda \wedge v_i \neq \lambda) \wedge \\ \exists k > 0, 1 \leq i_1 \leq n, \dots, 1 \leq i_k \leq n : (u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k})\}$$

Since nontermination is semi-decidable for shallow rewrite systems (Theorem [III](#)), we will reduce the restricted PCP to nontermination of shallow rewrite systems.

Theorem 5. *Termination of shallow rewrite systems is undecidable.*

Proof. Consider an instance $\langle u_1, v_1 \rangle \dots \langle u_n, v_n \rangle$ of the restricted PCP, that is, u_i, v_i are nonempty strings over alphabet Σ . We construct a shallow rewrite system R such that this PCP instance has a solution iff R is non-terminating.

Let $L = \text{Max}(|u_1|, \dots, |u_n|, |v_1|, \dots, |v_n|) + 2$. We construct R over a signature Σ' given by

$$\begin{aligned} \Sigma' &:= \Sigma'_0 \cup \Sigma'_1 \cup \Sigma'_2 \cup \Sigma'_6 \cup \Sigma'_8 \\ \Sigma'_1 &:= \Sigma \cup \{U_{i,j}, V_{i,j}, P_{i,j} : i = 1 \dots n, j = 1 \dots L\}, \\ \Sigma'_0 &:= \{U, U', V, V', P, P', P'', A, A', A''\} \\ \Sigma'_2 &:= \{f_1\}, \quad \Sigma'_6 := \{f_3\}, \quad \Sigma'_8 := \{f_2\} \end{aligned}$$

The j 'th symbol of u_i and v_i , whenever it exists, is denoted by $u_{i,j}$ and $v_{i,j}$ respectively. The rewrite system R is defined as follows:

$$\begin{aligned} R &:= R_U \cup R_V \cup R_{2P} \cup R_{P'} \cup R_{P''} \cup R_Q \cup R_\alpha \cup R_w \cup R_f \\ R_U &:= \{U_{i,1}U_{i,2} \dots U_{i,L}(U) \rightarrow U', U_{i,1}U_{i,2} \dots U_{i,L}(U') \rightarrow U' : i = 1, \dots, n\} \\ R_V &:= \{V_{i,1}V_{i,2} \dots V_{i,L}(V) \rightarrow V', V_{i,1}V_{i,2} \dots V_{i,L}(V') \rightarrow V' : i = 1, \dots, n\} \\ R_{2P} &:= \{U_{i,j}(x) \rightarrow P_{i,j}(x), V_{i,j}(x) \rightarrow P_{i,j}(x) : i = 1, \dots, n, j = 1, \dots, L\} \\ R_{P'} &:= \{P_{i,1}P_{i,2} \dots P_{i,L}(P') \rightarrow P' : i = 1, \dots, n\} \\ R_{P''} &:= \{P_{i,1}P_{i,2} \dots P_{i,L}(P'') \rightarrow P'' : i = 1, \dots, n\} \\ R_Q &:= \{U \rightarrow P, U \rightarrow A, V \rightarrow P, V \rightarrow A, A \rightarrow A', A \rightarrow A'', P \rightarrow P', P \rightarrow P''\} \\ R_\alpha &:= \{\alpha(A') \rightarrow A', \alpha(A'') \rightarrow A'' : \alpha \in \Sigma\} \\ R_w &:= \{U_{i,j}(x) \rightarrow u_{i,j}(x) : 1 \leq j \leq |u_i|\} \cup \{U_{i,j}(x) \rightarrow x : j > |u_i|\} \\ &\quad \cup \{V_{i,j}(x) \rightarrow v_{i,j}(x) : 1 \leq j \leq |v_i|\} \cup \{V_{i,j}(x) \rightarrow x : j > |v_i|\} \\ R_f &:= \{f_1(x, y) \rightarrow f_2(x, y, x, y, x, y, x, y), f_2(x, y, z, z, t, t, U', V') \rightarrow \\ &\quad f_3(x, y, z, z, t, t), f_3(x, y, A', A'', P', P'') \rightarrow f_1(x, y)\} \end{aligned}$$

\Rightarrow : We first show that if the PCP instance has a solution, then R is non-terminating. Let $w = u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$ be a solution of the PCP instance.

We show the existence of an infinite R -derivation starting from the ground term $s_1 := f_1(s_{uu}, s_{vv})$, where

$$\begin{aligned} s_{uu} &:= U_{i_1,1} \dots U_{i_1,L} \dots U_{i_k,1} \dots U_{i_k,L}(U) \\ s_{vv} &:= V_{i_1,1} \dots V_{i_1,L} \dots V_{i_k,1} \dots V_{i_k,L}(V) \end{aligned}$$

We rewrite s_1 using R_f to the term $s_2 := f_2(s_{uu}, s_{vv}, s_{uu}, s_{vv}, s_{uu}, s_{vv}, s_{uu}, s_{vv})$. Now, we do not touch positions 1 and 2 of s_2 . We rewrite position 3 using $R_w \cup \{U \rightarrow A\}$ to $w(A)$. We rewrite position 4 using $R_w \cup \{V \rightarrow A\}$ to $w(A)$. We rewrite position 5 with $R_{2P} \cup \{U \rightarrow P\}$ to $s_{pp} := P_{i_1,1} \dots P_{i_1,L} \dots P_{i_k,1} \dots P_{i_k,L}(P)$. Similarly, we rewrite position 6 with $R_{2P} \cup \{V \rightarrow P\}$ to s_{pp} . We rewrite positions 7 and 8 with R_U and R_V respectively to U' and V' . As a result, we reach the term $s_3 := f_2(s_{uu}, s_{vv}, w(A), w(A), s_{pp}, s_{pp}, U', V')$.

Now, we use R_f to rewrite s_3 to $s_4 := f_3(s_{uu}, s_{vv}, w(A), w(A), s_{pp}, s_{pp})$. Now we use $R_\alpha \cup \{A \rightarrow A'\}$ in position 3 to get A' . Similarly, we rewrite position 4 to A'' . In position 5, we use $R_{P'} \cup \{P \rightarrow P'\}$ to get P' . Similarly, we rewrite position 6 to P'' . This way we get the term $s_5 := f_3(s_{uu}, s_{vv}, A', A'', P', P'')$, which is rewritten by R_f to the starting term $s_1 := f_1(s_{uu}, s_{vv})$.

←: Suppose R does not terminate. We need to show that the PCP instance has a solution. To this end we define the concept of UV -variant. We say that a term s is a UV -variant of a term t , if t can be obtained from s by applying several rewrite steps using rules of the subset $\{U_{i,j}(x) \rightarrow x : i = 1 \dots n, j > |u_i|\} \cup \{V_{i,j}(x) \rightarrow x : i = 1 \dots n, j > |v_i|\}$ of R_w . Note that in this case, since the original PCP instance has not λ in the words of their pairs, s and t have the same number of occurrences of symbols of $\{U_{i,1} : i = 1 \dots n\} \cup \{V_{i,1} : i = 1 \dots n\}$.

Now, note that since all rules in R are height-preserving or decreasing, there is an infinite derivation with infinite rewrite steps at the top. We pick such an infinite derivation, but with minimal height initial term t . Then, the head of t has to be one of the f_i 's. Otherwise, no infinite steps can be done at the top preserving the height. Therefore, we have an infinite sequence $f_1(\dots) \rightarrow^* f_2(\dots) \rightarrow^* f_3(\dots) \rightarrow^* f_1(\dots) \rightarrow^* \dots$ at the top. We can assume that we start with a term of the form $f_1(u, v)$. By observing the R_f rules, one can deduce that u and v reach A', A'', P', P'' , and that u reaches U' and that v reaches V' . This is possible only if the terms u and v are UV -variants of terms of the form

$$\begin{aligned} s_{uu} &:= U_{i_1,1} \dots U_{i_1,L} \dots U_{i_k,1} \dots U_{i_k,L}(U) \\ s_{vv} &:= V_{j_1,1} \dots V_{j_1,L} \dots V_{j_{k'},1} \dots V_{j_{k'},L}(V) \end{aligned}$$

But, moreover, these terms have to be joinable to a term of the form $P_{i_1,1} \dots P_{i_k,L}(P)$, and also of the form $P_{j_1,1} \dots P_{j_{k'},L}(P)$. (Note here that since u_i, v_i are not λ , $U_{i,1} \dots U_{i,L}(x)$ can not rewrite to x and hence the indices $i_1, \dots, i_k, j_1, \dots, j_{k'}$ will be preserved in any joinability proof.) Hence, $k = k'$ and $i_r = j_r$ for all r . But moreover, u and v have to be joinable to a term of the form $u_{i_1,1} \dots u_{i_k,L}(A) = v_{i_1,1} \dots v_{i_k,L}(A)$. Hence, $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$ and there is a solution of the original PCP. \square

Theorem 6. *Termination and innermost termination of right-shallow rewrite systems is undecidable.*

Proof. Given an instance $\{(u_i, v_i) : i = 1, \dots, n\}$ of Post correspondence problem, we generate the rewrite system $R = \{f(x) \rightarrow g(x, x, x), g(x, u_i(y), v_i(z)) \rightarrow h(x, y, z), h(x, u_i(y), v_i(z)) \rightarrow h(x, y, z), h(x, \epsilon, \epsilon) \rightarrow f(x)\}$. Here ϵ is a constant representing the empty string. Note that R is right-shallow. It is easy to see that the PCP instance has a solution iff R is (innermost) non-terminating.

We remark here that the recently published proofs of undecidability of reachability, joinability and confluence of flat systems also use a reduction from PCP [8]. However, there are important differences and the construction in [8] can not be used directly in the proof of Theorem 5.

8 Conclusions

In this paper we showed that, given a right shallow rewrite system and a term, the existence of a (innermost) nonterminating derivation starting from the given term is decidable. Using this general result, we showed that innermost termination is decidable for shallow rewrite systems. We also used the same result to show decidability of general and innermost termination for right shallow-linear rewrite systems. We demonstrated that dropping assumptions on the rewrite systems leads to undecidability. In particular, general termination for shallow rewrite systems is undecidable and both general and innermost termination for right shallow rewrite systems is undecidable. As a result, we have narrowed the gap between decidability and undecidability of termination of rewrite systems. As further work it would be interesting to fix the exact complexity of these problems, but also to consider other classes of TRS, for example, basing its definition in its corresponding set of dependency pairs, like in [13].

References

1. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
2. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. ACM SIGPLAN 2006 Conf. Prog. Lang. Design and Impl. PLDI, pp. 415–426. ACM, New York (2006)
3. Dershowitz, N.: Termination of linear rewriting systems. In: Even, S., Kariv, O. (eds.) Automata, Languages and Programming. LNCS, vol. 115, pp. 448–458. Springer, Heidelberg (1981)
4. Godoy, G., Tiwari, A.: Deciding fundamental properties of right-(ground or variable) rewrite systems by rewrite closure. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 91–106. Springer, Heidelberg (2004)
5. Godoy, G., Tiwari, A.: Termination of rewrite systems with shallow right-linear, collapsing, and right-ground rules. In: Nieuwenhuis, R. (ed.) Automated Deduction – CADE-20. LNCS (LNAI), vol. 3632, pp. 164–176. Springer, Heidelberg (2005)
6. Hart, S., Sharir, M., Pnueli, A.: Termination of probabilistic concurrent program. ACM Trans. Program. Lang. Syst. 5(3), 356–380 (1983)

7. Huet, G., Lankford, D.S.: On the uniform halting problem for term rewriting systems. INRIA, Le Chesnay, France, Technical Report 283 (1978)
8. Mitsuhashi, I., Oyamaguchi, M., Jacquemard, F.: The confluence problem for flat TRSs. In: Calmet, J., Ida, T., Wang, D. (eds.) AISC 2006. LNCS (LNAI), vol. 4120, pp. 68–81. Springer, Heidelberg (2006)
9. Plaisted, D.A.: Polynomial time termination and constraint satisfaction tests. In: Kirchner, C. (ed.) Rewriting Techniques and Applications. LNCS, vol. 690, pp. 405–420. Springer, Heidelberg (1993)
10. Takai, T., Kaji, Y., Seki, H.: Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 246–260. Springer, Heidelberg (2000)
11. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004)
12. Toyama, Y.: Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters* 25, 141–143 (1987)
13. Wang, Y., Sakai, M.: Decidability of termination for semi-constructor trss, left-linear shallow trss and related systems. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 343–356. Springer, Heidelberg (2006)

Abstract Critical Pairs and Confluence of Arbitrary Binary Relations

Rémy Haemmerlé and François Fages

Projet Contraintes – INRIA Rocquencourt – France

FirstName.LastName@inria.fr

Abstract. In a seminal paper, Huet introduced abstract properties of term rewriting systems, and the confluence analysis of terminating term rewriting systems by critical pairs computation. In this paper, we provide an abstract notion of critical pair for arbitrary binary relations and context operators. We show how this notion applies to the confluence analysis of various transition systems, ranging from classical term rewriting systems to production rules with constraints and partial control strategies, such as the Constraint Handling Rules language CHR. Interestingly, we show in all these cases that some classical critical pairs can be disregarded. The crux of these analyses is the ability to compute critical pairs between states built with general context operators, on which a bounded, not necessarily well-founded, ordering is assumed.

Dedicated to Gérard Huet on his 60th birthday.

1 Introduction

In a seminal paper [9], Huet introduced abstract properties of term rewriting systems, and the confluence analysis of terminating term rewriting systems by critical pairs computation. Since then, the notion of critical pairs obtained by superposing the left-hand sides of rewriting rules has been applied to a wide variety of rewriting systems, ranging from pure term rewriting systems (TRS), to TRS in equational theories [14], conditional TRS [7], rewriting models of concurrency [13], rewriting logic [12,4], higher-order rewriting [2] and graph rewriting [5,15]. Similarly, Knuth-Bendix like procedures [14,10,16] for completing non confluent rewriting systems into confluent rewriting systems have been generalized to these different settings.

To date however, although the notion of critical pairs has been adapted to a variety of formalisms, there is no general definition of an abstract notion of critical pairs from which the concrete definitions could be obtained as particular instances. In the categorical formulations of rewriting systems, the notion of relative pushouts [11] does provide an abstract condition for contextual equivalences but we are not aware of an abstract critical pair lemma in the categorical setting. Recently, in the framework of canonical inference [3,6], notions of criticality and completions have been developed for abstract proof systems, assuming

a well-founded ordering on proofs. However, we will show that this assumption is too strong for the confluence analysis of binary relations in dense structures, and will illustrate this situation with a concrete example.

In this paper, we provide an abstract notion of critical pair for arbitrary binary relations and context operators. We show how this notion applies to the confluence analysis of term rewriting systems, conditional TRS, and production rules with constraints such as the Constraint Handling Rules language CHR [8], under both its naive semantics and its refined semantics with partial control structures [1]. The crux of these analyses is the ability to compute critical pairs between states built with general context operators, on which a bounded, not necessarily well-founded, ordering is assumed.

The next section gives some preliminary notations about binary relations and their composition. In section 3 we propose some abstract counterparts of the notions of context, context compatibility and substitution stability from [9], and prove an abstract critical pair theorem for establishing the confluence of arbitrary binary relations. In section 4 we show how our abstract definitions can be instantiated to prove the soundness of the classical notions of critical pairs in ordinary TRS and conditional TRS. In section 5 we proceed similarly to show the soundness of the classical definitions of critical pairs in CHR, respectively under its naive semantics and under its refined semantics that includes a partial control strategy stating that a rule is fired only once on the same instances [1].

Interestingly, we show in all these cases that some classical critical pairs can be disregarded. We conclude on the generality of this work, and on some perspectives for future work.

2 Preliminaries

Let E be an arbitrary set and $\rightarrow \subset E \times E$ be an arbitrary relation on E , called here *reduction*. We shall use the following notations and definitions:

- $i = \{e \rightarrow e \mid e \in E\}$ is the identity relation on E ;
- \circ is the composition : $\rightarrow_a \circ \rightarrow_b = \{(e_1, e_2) \mid \exists e \in E (e_1 \rightarrow_a e \wedge e \rightarrow_b e_2)\}$;
- $\rightarrow^{-1} = \{(e_2, e_1) \mid (e_1 \rightarrow e_2)\}$ is the inverse relation of \rightarrow ;
- $\rightarrow^0 = i$ and $\rightarrow^n = \rightarrow \circ \rightarrow^{n-1}$ for $n \geq 1$;
- $\rightarrow^* = \bigcup_{i \geq 0} \rightarrow^i$, the transitive-reflexive closure of \rightarrow ;
- $\rightarrow_\epsilon = \rightarrow \cup i$;
- $\uparrow = \rightarrow^* \circ \rightarrow^{-1}$, the *common ancestor* relation ;
- $\downarrow = \rightarrow^{-1} \circ \rightarrow$, the *common direct ancestor* relation.
- $\downarrow_\epsilon = \rightarrow^* \circ \rightarrow_\epsilon^{-1}$ the *common descendent* relation ;
- $\downarrow_\epsilon = \rightarrow_\epsilon \circ \rightarrow_\epsilon^{-1}$.
- \rightarrow is *noetherian* if there is no infinite sequence $e_0 \rightarrow e_1 \rightarrow \dots$
- \rightarrow is *confluent* if $\forall e_1, e_2 \in E (e_1 \uparrow e_2 \Rightarrow e_1 \downarrow e_2)$.
- \rightarrow is *locally confluent* if $\forall e_1, e_2 \in E (e_1 \downarrow_\epsilon e_2 \Rightarrow e_1 \downarrow e_2)$.
- \rightarrow is *strongly confluent* if $\forall e_1, e_2 \in E (e_1 \downarrow_\epsilon e_2 \Rightarrow e_1 \downarrow_\epsilon e_2)$.

Obviously, strong confluence implies confluence, and by Newman's lemma we know that a noetherian reduction relation is confluent if and only if it is locally confluent [9]. A pair (e_1, e_2) of elements in E is \rightarrow -joinable if $e_1 \downarrow e_2$, and \rightarrow -strongly joinable if $e \downarrow_\epsilon e_2$. A set of pairs will be said \rightarrow -joinable if all its pairs are \rightarrow -joinable.

3 Abstract Critical Pairs

3.1 Abstract Contexts

In all this section, a binary relation $\rightarrow \subset E \times E$ is assumed. We provide an abstract counterpart of the notions of contexts, context compatibility and substitution stability, introduced in [9] for TRS. To this end, we study families of operators on E that generalize the operations of putting a term in a context or instantiating a term by a substitution.

Definition 1 (\rightarrow -compatible operators). *An n -ary operator $C : E^n \rightarrow E$ is \rightarrow -compatible if $C(e_1, \dots, e_n) \xrightarrow{*} C(e_1, \dots, e_{i-1}, e', e_{i+1}, \dots, e_n)$ whenever $e_i \rightarrow e'$ for any index i , $1 \leq i \leq n$.*

Proposition 1. *An n -ary operator C over E is \rightarrow -compatible if and only if $C(e_1, \dots, e_n) \xrightarrow{*} C(e'_1, \dots, e'_n)$ whenever $e_i \xrightarrow{*} e'_i$ for all i , $1 \leq i \leq n$.*

Proposition 2

- (i) *The composition of \rightarrow -compatible operators is \rightarrow -compatible.*
- (ii) *The projection $\pi_i^n = \lambda x_1 \dots x_n. x_i$ (with $1 \leq i \leq n$) is \rightarrow -compatible.*
- (iii) *Permuting the arguments of an operator preserves its compatibility.*

Proof To prove (i) let us suppose that C_1 and C_2 are two \rightarrow -compatible operators over E of arity n_1 and n_2 respectively. Let $n = n_1 + n_2 - 1$ and let us suppose $e_i \xrightarrow{*} e'_i$ for all $1 \leq i \leq n$. We have $C_1(e_i, \dots, e_{i+n_1}) \xrightarrow{*} C_1(e'_i, \dots, e'_{i+n_1})$ by the previous proposition, for any i , $1 \leq i \leq n_2$. Furthermore we have $C_2(e_1, \dots, e_{i-1}, C_1(e_i, \dots, e_{i+n_1}), e_{i+n_1+1}, \dots, e_{n_2}) \xrightarrow{*} C_2(e'_1, \dots, e'_{i-1}, C_1(e'_i, \dots, e'_{i+n_1}), e'_{i+n_1+1}, \dots, e'_{n_2})$ since \rightarrow and the identity relation are included in $\xrightarrow{*}$. Hence the composition of C_1 and C_2 is \rightarrow -compatible. For (ii) and (iii), the \rightarrow -compatibility of the projection operators and of any \rightarrow -compatible operator with a permutation of its arguments follows directly from the definition. \square

Definition 2 (\rightarrow -Contexts). *A family of \rightarrow -contexts is a family of \rightarrow -compatible operators containing E (as constant operators) and closed by projection, composition and argument permutation. We will denote by \mathcal{C}^n the set of n -ary contexts of \mathcal{C} , for $n \geq 0$.*

3.2 Abstract Linear Contexts

Definition 3 (Linear \rightarrow -contexts). *An n -ary \rightarrow -context C is linear if whenever $e_i \rightarrow e'$ then $C(e_1, \dots, e_n) \xrightarrow{\epsilon} C(e_1, \dots, e_{i-1}, e', e_{i+1}, \dots, e_n)$.*

A linear \rightarrow -context is obviously \rightarrow -compatible. Furthermore, we have :

Proposition 3. *The composition of linear contexts is linear. The projections are linear contexts. Permuting the arguments of a context preserves its linearity.*

Now, let us denote by e^n the sequence of n repetitions of the element e .

Definition 4 (Absorbing \rightarrow -contexts). *An n -ary \rightarrow -context C is absorbing if there exists an index i , $1 \leq i \leq n$, such that $\forall e_0, e_1, \dots, e_n \in E$ $C(e_1, \dots, e_n) = C(e_1, \dots, e_{i-1}, e_0, e_{i+1}, \dots, e_n)$.*

Definition 5 (Linear decomposition of \rightarrow -contexts). *A linear n -ary \rightarrow -context C is a linear decomposition of an unary \rightarrow -context C' if for any element $e \in E$ we have $C'(e) = C(e^n)$. A family of linear \rightarrow -contexts is linear if any linear decomposition of its unary contexts is either unary or absorbing.*

3.3 \mathcal{C} -Safe Pairs

A family \mathcal{C} of \rightarrow -contexts induces a preordering relation over pairs of elements in E as follows :

Definition 6. *The preorder induced by a family \mathcal{C} of contexts is the relation $\geq_{\mathcal{C}}$ on pairs satisfying: $(e'_1, e'_2) \geq_{\mathcal{C}} (e_1, e_2) \Leftrightarrow \exists C \in \mathcal{C}. (e'_1 = C(e_1) \wedge e'_2 = C(e_2))$.*

Proposition 4. *$(E, \geq_{\mathcal{C}})$ is a preorder.*

Proof. The reflexivity of $\geq_{\mathcal{C}}$ follows from the fact that the projection π_1^1 is in \mathcal{C} . The transitivity of $\geq_{\mathcal{C}}$ follows from the closure of contexts under arbitrary compositions. □

In the following, we will denote by $>_{\mathcal{C}}$ the strict preorder associated to $\geq_{\mathcal{C}}$. In this preorder, the joinability of a pair entails the joinability of all its $\geq_{\mathcal{C}}$ -greater pairs :

Lemma 1. *Let \mathcal{C} be a family of \rightarrow -contexts and (e_1, e_2) and (e'_1, e'_2) be two pairs in E such that $(e'_1, e'_2) \geq_{\mathcal{C}} (e_1, e_2)$. (i) If $e_1 \downarrow e_2$ then $e'_1 \downarrow e'_2$. Furthermore (ii), if \mathcal{C} is linear and $e_1 \downarrow_{\epsilon} e_2$ then $e'_1 \downarrow_{\epsilon} e'_2$.*

Proof. Since $(e'_1, e'_2) \geq_{\mathcal{C}} (e_1, e_2)$, let $C \in \mathcal{C}$ be a context such that $e'_1 = C(e_1)$ and $e'_2 = C(e_2)$. Concerning (i), as $e_1 \downarrow e_2$, there exists an e such that $e_1 \xrightarrow{*} e$ and $e_2 \xrightarrow{*} e$. Hence by proposition □ we have $C(e_1) \xrightarrow{*} C(e)$ and $C(e_2) \xrightarrow{*} C(e)$, hence $e'_1 \downarrow e'_2$. (ii) is a direct consequence of the linearity of the contexts in \mathcal{C} . □

One can also remark that a symmetrical pair is greater than any other pair thanks to the projection operators in \mathcal{C} . The joinability of symmetrical pairs is thus subsumed by the joinability of non symmetrical pairs. We call \mathcal{C} -Safe pairs those pairs that are joinable by the \rightarrow -compatibility of contexts. \mathcal{C} -safe pairs can thus be removed from the confluence analysis of $\bigwedge_{\mathcal{C}}$ pairs.

Definition 7 (*C-Safe Pairs*). A *C*-safe pair w.r.t. a context $C \in \mathcal{C}^2$ and two transitions $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ is a pair of the form $(C_1(s_1 \dots s_m), C_2(t_1 \dots t_n))$ such that:

- for all $1 \leq i \leq m$, $s_i \in \{l_2, r_2\}$ and for all $1 \leq j \leq m$, $t_j \in \{l_1, r_1\}$.
- $C_1 \in \mathcal{C}^m$ and $C_2 \in \mathcal{C}^n$ are linear decompositions of respectively $\lambda x.C(l_1, x)$ and $\lambda x.C(x, l_2)$.

Lemma 2. All *C*-safe pairs are joinable.

Proof. In a *C*-safe pair as in the definition above, we trivially have $t_i \xrightarrow{*} r_2$ for all $1 \leq i \leq m$, and $s_j \xrightarrow{*} r_1$ for all $1 \leq j \leq m$. Hence, by proposition [□](#), $C_1(t_1, \dots, t_m) \xrightarrow{*} C_1(r_2^m)$ and $C_2(s_1, \dots, s_n) \xrightarrow{*} C_2(r_1^n)$. Since C_1 and C_2 are the respective linear decompositions of $\lambda x.C(l_1, x)$ and $\lambda x.C(x, l_2)$ for some context C , we also have $C_1(r_2^m) = C(l_1, r_2)$ and $C_2(r_1^n) = C(r_1, l_2)$. The pairs are thus joinable by the \rightarrow -compatibility of C . □

Now, by considering the linear decomposition of \rightarrow -contexts, we get :

Lemma 3. Let \mathcal{C} be a linear family of contexts and C an unary context in \mathcal{C} . If $C_n \in \mathcal{C}^n$ is a linear decomposition of C then there exists $1 \leq i \leq n$ such that for all $e_1, \dots, e_n \in E$, $C_n(e_1, \dots, e_n) = C'(e_i)$.

Proof. The proof is by induction on n . The base case, where $n = 1$, is trivial. In the inductive case, where $n > 1$, since the family \mathcal{C} is linear, C_n is absorbing. Hence there exist i , $1 \leq j \leq n$, and $e' \in E$ such that for all $e \in E$ $C_n(e^n) = C_n(e^{(j-1)}, e', e^{(n-j-1)})$. The context $\lambda x_1, \dots, x_{n-1}.C_n(x_1, \dots, x_{j-1}, e', x_j \dots x_{n-1})$ is thus a linear decomposition of C' , and the induction hypothesis concludes the proof.

Proposition 5. Let \mathcal{C} be a linear family of \rightarrow -contexts. Any *C*-safe pair w.r.t. two transitions $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ is of the form $(C(l_1, r_2), C(r_1, l_2))$ for some binary \rightarrow -context $C \in \mathcal{C}^2$.

Proof. By lemma [3](#), any *C*-safe pair w.r.t. some context C' and transitions $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$, is of the form $(C'(l_1, s), C'(t, l_2))$ with $s \in \{l_2, r_2\}$ and $t \in \{l_1, r_1\}$, i.e. of the form : (i) $(C'(l_1, l_2), C'(l_1, l_2))$, (ii) $(C'(l_1, l_2), C'(r_1, l_2))$, (iii) $(C'(l_1, r_2), C'(l_1, l_2))$, or (iv) $(C'(l_1, r_2), C'(r_1, l_2))$. Hence the choice $C = \lambda x_1.x_2.\pi_1^3(x_1, x_2, C'(l_1, l_2))$ for (i), $C = \lambda x_1.x_2.\pi_1^2(C'(x_1, l_2), x_2)$ for (ii), $C = \lambda x_1.x_2.\pi_2^2(x_1, C'(l_1, x_2))$ for (iii) or $C = \lambda x_1.x_2.\pi_1^3(x_1, x_2, C'(l_1, l_2))$ for (iv) respectively, ends the proof. □

Lemma 4. If \mathcal{C} is a linear family of \rightarrow -context, then all *C*-safe pairs are strongly joinable.

Proof. By the previous proposition, we know that any *C*-safe pair is of the form $(C(l_1, r_2), C(r_1, l_2))$ with $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$. Hence, by the linearity of C we have $C(l_1, r_2) \xrightarrow{\hookrightarrow} C(r_1, r_2)$ and $C(r_1, l_2) \xrightarrow{\hookrightarrow} C(r_1, r_2)$. □

3.4 C-Critical Pairs

Let us recall that an ordered set (E, \geq) is *lower bounded* if any element of E is greater than or equal to some minimal element of E .

Definition 8 (C-critical pair). A C-critical pair is a \geq_C -minimal element of (Δ) that is not C-safe.

Theorem 1. Let \mathcal{C} be a family of \rightarrow -contexts (resp. family of linear \rightarrow -contexts) such that (Δ) is lower bounded w.r.t. \geq_C . \rightarrow is locally confluent (resp. strongly confluent) if and only if all C-critical pairs are joinable (resp. strongly joinable).

Proof. For the *if* direction, let us suppose that all C-critical pairs are joinable. We know that any pair in (Δ) is either a C-safe pair, or is comparable to a C-critical pair. It is thus joinable in both cases, using respectively lemma 2 and lemma 1. The “only if” is trivial since C-critical pairs are in (Δ) .

The proof of strong confluence is similar using lemma 4. □

This theorem can be used to establish the local (resp. strong) confluence of an arbitrary binary relation \rightarrow , by finding a family \mathcal{C} of (linear) \rightarrow -context operators for which \geq_C is lower bounded on brother pairs and admits a finite set of \geq_C -minimal elements. In the following sections, we illustrate this approach on several examples. We also show that some classical critical pairs are not C-critical, and can thus be disregarded. Furthermore, one example in section 5.2 shows a concrete case where the preordering \geq_C is not well-founded but only lower bounded on brother pairs, as the theorem requires.

4 Applications to Term Rewriting Systems

4.1 Preliminaries

Let \mathcal{T} be the set of *terms* (denoted by t, s, r, \dots) built from a countable infinite set \mathcal{V} of variables (denoted by x, y, z, \dots) and a countable set \mathcal{F} of function symbols (denoted by f, g, h, \dots) given with their arity. We will use the classical propositions and notations borrowed from 9 :

- $\mathcal{V}(t) \subset \mathcal{V}$ for the set of variables of t ,
- $\mathcal{O}(t)$ for the set of occurrences of t (denoted by u, v, \dots),
- t/u for the subterm of t at u ,
- $t[u \leftarrow s]$ for the subterm replacement at u ,
- $u.v$ for the concatenation of u and v ,
- \leq the prefix ordering on occurrences,
- $u|v$ to note disjoint occurrences.

A *substitution* σ is mapping from \mathcal{V} to \mathcal{T} with $x\sigma = x$ almost everywhere. Substitution are extended as morphisms to \mathcal{T} . If $\{x_1, \dots, x_n\}$ is the *domain* of σ (i.e. the set $\{x \in \mathcal{V} | \sigma(x) \neq x\}$) we will also denote σ by $[x_1 \setminus x_1\sigma, \dots, x_n \setminus x_n\sigma]$. A *renaming* is a substitution $[x_1 \setminus y_1, \dots, x_n \setminus y_n]$ where the y_i 's are pairwise distinct variables. A *most general unifier* (mgu) for two given terms t and s is

a substitution σ satisfying (i) $t\sigma = s\sigma$ and (ii) for all substitution σ_1 if $t\sigma_1 = s\sigma_1$ then there exists a substitution σ_2 such that $\sigma_1 = \sigma_2 \circ \sigma$.

A *rewriting rule* is a pair $\langle l \rightarrow r \rangle$ where l and r are two first order terms such that $l \notin \mathcal{V}$ and $\mathcal{V}(r) \subset \mathcal{V}(l)$. A *term rewriting system* \mathcal{R} is a set of rewriting rules. The relation $\rightarrow_{\mathcal{R}}$ is the least relation satisfying $s \rightarrow_{\mathcal{R}} t$ if there exists a position p , a rule $\langle l \rightarrow r \rangle \in \mathcal{R}$ and a substitution σ such that $s/u = l\sigma$ and $t = s[u \leftarrow r\sigma]$.

4.2 \mathcal{C} -critical Pairs of Ordinary Term Rewriting Systems

Let \mathcal{R} be a TRS over a set of terms \mathcal{T} .

Definition 9. Let \mathcal{C}_t be the family of operators over \mathcal{T} containing all the operators of the form $C_{y_1, \dots, y_n} = \lambda s, s_1, \dots, s_n . s[y_1 \setminus s_1, \dots, y_n \setminus s_n]$ for $\{y_1, \dots, y_n\} \subset \mathcal{V}$, and closed under composition, projection and argument permutation.

One can easily show :

Proposition 6. \mathcal{C}_t is a family of $\rightarrow_{\mathcal{R}}$ -contexts.

Proposition 7. $\geq_{\mathcal{C}_t}$ is well-founded.

Therefore all subsets of $(E \times E)$, and in particular (Δ) , are lower bounded. Now a *classical critical pair*, between two rules $\langle l_i \rightarrow r_i \rangle$ and $\langle l_j \rightarrow r_j \rangle$ is a pair of the form $(l_i\sigma[u \leftarrow r_j\rho\sigma], r_i\sigma)$ where :

1. ρ is a renaming and σ is a substitution such that $\mathcal{V}(l_i) \cap \mathcal{V}(l_j\rho) = \emptyset$;
2. u is an occurrence of l_i such that l_i/u is not a variable;
3. σ is an mgu for l_i/u and $l_j\rho$.

Proposition 8. Let t be an arbitrary term and y a variable. (i) $\lambda s.C_y(s, t)$ is a linear $\rightarrow_{\mathcal{R}}$ -context. (ii) $\lambda s_1 \dots s_n.C_{y_1, \dots, y_n}(t[u_1 \leftarrow y_1] \dots [u_n \leftarrow y_n], s_1, \dots, s_n)$, where the u_i 's are the occurrences of y in t and y_i 's are pairwise distinct variables free in t , is a linear decompositions of $\lambda s.C_y(t, s)$.

Then by tedious case analysis, one can show :

Proposition 9. Let \mathcal{R} be a rewriting system. A \mathcal{C}_t -critical pair of $\rightarrow_{\mathcal{R}}$ is a classical critical pair between two rules of \mathcal{R} .

This proposition together with theorem [II](#) establishes that the local confluence of arbitrary rewriting systems can be deduced from the joinability of its classical critical pairs. However, some classical critical pairs may be not \mathcal{C}_t -critical, and can thus be disregarded.

Example 1. Let \mathcal{R} be the following system:

$$\mathcal{R} = \{ \langle a \rightarrow f(c, b) \rangle, \langle a \rightarrow g(b) \rangle, \langle f(x, x) \rightarrow g(x) \rangle, \langle b \rightarrow c \rangle \}$$

$(f(c, b), g(b))$ is a classical critical pair for $\rightarrow_{\mathcal{R}}$. However this critical pair is a \mathcal{C}_t -safe pair w.r.t. the transitions $f(x, x) \rightarrow g(x)$ and $b \rightarrow c$. Indeed $C_1 = \lambda s_1, s_2.f(s_1, s_2)$ is a linear decomposition of $\lambda s.C_x(g(x), s)$, $C_2 = \lambda s.C_x(s, b)$ is linear, $f(c, b) = C_1(c, b)$ and $g(b) = C_2(g(x))$ with $c \rightarrow b$ and $f(x, x) \rightarrow g(x)$.

4.3 Strong Confluence of Linear Term Rewriting Systems

A term t is *linear* if any variable appears at most once in t . A system \mathcal{R} is *linear* iff for all rules $\langle l \rightarrow r \rangle \in \mathcal{R}$, l and r are linear. Let $\rightarrow_{\mathcal{R}}^l$ be the restriction of $\rightarrow_{\mathcal{R}}$ to linear terms.

Definition 10. Let \mathcal{C}_l be the family of contexts, closed by composition, argument permutation, projection and containing the operators

$$C_{y_1, \dots, y_n}^{\rho} = \lambda s_0, \dots, s_n. (\phi_0(s_0)[y_1 \setminus \phi_1(s_1), \dots, y_n \setminus \phi_n(s_n)])\rho$$

where y_1, \dots, y_n are pairwise distinct variables, ρ is a renaming, $\cup_{i \in \mathbb{N}} \{\mathcal{V}_i\}$ is a partition of \mathcal{V} where the \mathcal{V}_i 's are infinite, and for any $i \in \mathbb{N}$, ϕ_i is a one-to-one mapping between \mathcal{V} and \mathcal{V}_i .

Let \mathcal{C}_s be the family of operators containing all substitution operators, closed by projection, argument permutation and composition.

Proposition 10. For any relation a linear TRS \mathcal{R} we have :

- (i) \mathcal{C}_l and \mathcal{C}_s are linear families of respectively $\rightarrow_{\mathcal{R}}^l$ -contexts and $\rightarrow_{\mathcal{R}}$ -contexts;
- (ii) the brother pairs w.r.t. $\rightarrow_{\mathcal{R}}^l$ are lower bounded w.r.t. $\geq_{\mathcal{C}_l}$;
- (iii) \mathcal{C}_l -critical pairs of $\rightarrow_{\mathcal{R}}^l$ are classical critical pairs;
- (iv) Any $\rightarrow_{\mathcal{R}}$ -brother pair is greater than or equal to a $\rightarrow_{\mathcal{R}}^l$ -brother pair.

The propositions (i), (ii) and (iii) can be used with theorem [11](#) to analyze the strong confluence of $\rightarrow_{\mathcal{R}}^l$. The strong confluence analysis of any linear rewriting system follows from (i), (iv) and lemma [11](#).

4.4 Conditional Term Rewriting Systems

A Conditional Term Rewriting Systems (CTRS) is a TRS in which the application of rules is controlled by some condition. In this section we focus on particular CRTS known as *join systems* [\[16\]](#).

A *conditional term rewriting rule* has the form $\langle l \rightarrow r \Leftarrow t \downarrow t' \rangle$ where $\langle l \rightarrow r \rangle$ is a classical rewriting rule and t and t' are terms. A *conditional term rewriting system* (CTRS) is a set of conditional term rewriting rules. For a given CTRS \mathcal{R}_C , the relation $\rightarrow_{\mathcal{R}_C}$ is defined inductively by the following rule :

$$\frac{u \in \mathcal{O}(s) \quad \langle l \rightarrow r \Leftarrow t \downarrow t' \rangle \in \mathcal{R}_C \quad t\sigma \downarrow_{\mathcal{R}_C} t'\sigma}{s[u \leftarrow l\sigma] \rightarrow_{\mathcal{R}_C} s[u \leftarrow r\sigma]}$$

Definition 11. A primary critical pair between two conditional rules $\langle l_i \rightarrow r_i \Leftarrow t_i \downarrow t'_i \rangle$ and $\langle l_j \rightarrow r_j \Leftarrow t_j \downarrow t'_j \rangle$ is a conditioned pair of the form $((t_i\sigma, t'_j\rho\sigma) \downarrow (t'_i\sigma, t'_j\rho\sigma) : (l_i\sigma[u \leftarrow r_j\rho\sigma], r_i\sigma))$ where:

- ρ is a renaming and σ is a substitution such that $\mathcal{V}(l_i) \cap \mathcal{V}(l_j\rho) = \emptyset$;
- u is an occurrence of l_i such that l_i/u is not a variable;
- σ is an mgu for l_i/u and $l_j\rho$;

A conditioned pair $(t \downarrow t' : (s, s'))$ is joinable if $s\sigma \downarrow s'\sigma$ for any substitution σ such that $t\sigma \downarrow t'\sigma$.

This definition of critical pair was proposed in [7]. Nonetheless the authors of this paper underlined that there exist noetherian non locally confluent systems for which all those critical pairs are joinable. The following example illustrates why the previous definition is not consistent with our definition of abstract critical pairs. We then introduce the definition of secondary critical pairs.

Example 2. Let $\mathcal{R}_C = \{(f(x) \rightarrow g(x) \Leftarrow x \downarrow a), (a \rightarrow b \Leftarrow a \downarrow a)\}$. The family of contexts \mathcal{C}_t , defined in section 4.2 for ordinary TRS, is a family of contexts for any $\rightarrow_{\mathcal{R}_C}$. Nonetheless, since $f(x) \not\rightarrow_{\mathcal{R}_C} g(x)$, the pair $(f(b), g(a))$ is not \mathcal{C}_t -safe. Furthermore, as it is minimal in (Δ) , it is \mathcal{C}_t -critical and should be considered.

Definition 12. A secondary critical pair between two conditional rules $(l_i \rightarrow r_i \Leftarrow t_i \downarrow t'_i)$ and $(l_j \rightarrow r_j \Leftarrow t_j \downarrow t'_j)$ is a pair of the form $((t_i, t_j\rho) \downarrow (t'_i, t'_j\rho) : (l_i[x \setminus u[p \leftarrow r_j\rho]], r_i[x \setminus u[p \leftarrow l_j\rho]])$ where:

- ρ is a renaming such that $\mathcal{V}(l_i) \cap \mathcal{V}(l_j\rho) = \emptyset$;
- u is an arbitrary term and $p \in \mathcal{O}(u)$;
- $x \in \mathcal{V}(l_i) \cap \mathcal{V}(t_i \downarrow t'_i)$.

According to the definition of joinability of conditioned pairs, a conditioned pair $(t \downarrow t' : (s, s'))$ defines the set of pairs $\{(s\sigma, s'\sigma) \mid t\sigma \downarrow t'\sigma\}$. Let us call an *instance* of $(t \downarrow t' : (s, s'))$, any element of this set. With a proof analogous to proposition 9 we obtain :

Proposition 11. Let \mathcal{R}_C be a rewriting system. A \mathcal{C}_t -critical pair of $\rightarrow_{\mathcal{R}_C}$ is an instance of a primary or secondary critical pair between two rules of \mathcal{R}_C .

This shows the soundness of deriving the local confluence of CTRS from the joinability of both primary and secondary critical pairs. For this, an effective definition of secondary critical pairs is thus worth investigating.

5 Applications to Production Rules with Constraints

Production rules are condition-action rules that transform a base of facts by adding or removing facts at each rule firing. The Constraint Handling Rules (CHR) language [8] generalizes production rules by lifting the base of ground facts to a store of constraints over uninstantiated variables, and interpreted in an arbitrary mathematical structure.

In this section, we focus on the confluence analysis of CHR rules, and show how the abstract notion of \mathcal{C} -critical pairs can be instantiated to analyze the confluence of CHR rules as proposed in [1]. This is shown under both the naive semantics of CHR without control strategy, and under the refined semantics of CHR that integrates partial control strategies based on the history on rule firings, captured here by context operators. Furthermore, the necessity to deal with constrained states illustrates the difficulty to define well-founded orderings, and our use of bounded orderings instead.

5.1 Preliminaries

In CHR, a language of *built-in constraints* interpreted over some structure \mathcal{X} and assumed to contain the equality $=$, is distinguished from the language of user-defined *CHR constraints* formed over a different set of predicate symbols. A CHR program is a finite sequence of CHR rules, where a CHR rule is either:

- a *simplification* rule of the form:
 $H_1, \dots, H_i \iff G_1, \dots, G_j \mid B_1, \dots, B_k$
- or a *propagation* rule of the form:
 $H_1, \dots, H_i \implies G_1, \dots, G_j \mid B_1, \dots, B_k$

where $i > 0, j \geq 0, k \geq 0, l > 0, H_1, \dots, H_i$ are CHR constraints, the guards G_1, \dots, G_j are built-in constraints, and the body B_1, \dots, B_k is composed of CHR and built-in constraints (with $k > 0$ in propagation rules).

The symbol \top is used to represent empty sequences. The empty guard can be omitted together with the symbol \mid . The notation *name*@*R* gives a name to a CHR rule *R*. For the sake of simplicity, we assume without loss of generality that a variable appears at most once in the head of a rule.

Example 3. The following CHR rules [8] define an ordering constraint solver

```

reflexivity @ X=<Y <=> X=Y | true.
antisymmetry @ X=<Y , Y=<X <=> X=Y.
transitivity @ W=<X , Y=<Z ==> X=Y | W=<Z.
    
```

The first rule eliminates the $=<$ constraints with equal arguments, the second replaces a double inequality by an equality, and the third adds the transitive closure constraints.

5.2 CHR Under Its Naive Semantics

The naive operational semantics of CHR does not include any control strategy. As a result, propagation rules can loop forever. This is corrected in the refined semantics presented in the next section by imposing that a rule is fired once on the same instances. We first present the confluence analysis of CHR programs under the naive semantics.

Here, a *CHR state* is a tuple $\exists \bar{x}. \langle F, E, D \rangle$ where, \bar{x} is a set of variables called *anonymous variables*, F is a multiset of built-in and CHR constraints called *goal*, E is a CHR constraint store, and D is a built-in constraint store. A state is thus a conjunction of CHR and built-in constraints [4]. In the following, we work implicitly modulo the following equivalence \equiv over states :

- 1 $\exists \bar{x}y. \langle F, E, D \rangle \equiv \exists \bar{x}z. (\langle F, E, D \rangle [y \setminus z])$ with $zy \notin \bar{x}$.
- 2 $\exists \bar{x}\bar{y}. \langle F, E, D \rangle \equiv \exists \bar{x}\bar{y}'. \langle F, E, D' \rangle$ if $\mathcal{X} \models \exists \bar{y}. D \Leftrightarrow \exists \bar{y}'. D'$ if $(\bar{y} \cup \bar{y}') \cap \mathcal{V}(F, E) = \emptyset$

¹ Usually a CHR goal is annotated with the free variables of the query (i.e. initial goal). Here, the anonymous variables represent the variables introduced during the computation that leads to the given state.

The condition at the end of the second rule ensures that the variables \bar{y} and \bar{y}' are *strictly local variables*, i.e. anonymous variables appearing only in the built-in store. Given a CHR program P , the transition relation \rightarrow over states of the naive operational semantics, is defined inductively as the least relation satisfying the following rules :

Solve $\exists \bar{x}. \langle C \wedge F, E, D \rangle \rightarrow \exists \bar{x}. \langle F, E, C \wedge D \rangle$ if C is a built-in constraint

Introduce $\exists \bar{x}. \langle H \wedge F, E, D \rangle \rightarrow \exists \bar{x}. \langle F, H \wedge E, D \rangle$ if H is a CHR constraint.

Simplify $\exists \bar{x}. \langle F, H' \wedge E, D \rangle \rightarrow \exists \bar{x}\bar{y}. \langle B \wedge F, E, H = H' \wedge D \rangle$
 if $(H \Leftrightarrow G \mid B)$ is in P renamed with fresh variables \bar{y} ,
 and $\mathcal{X} \models D \rightarrow \exists \bar{y}(H = H' \wedge G)$.

Propagate $\exists \bar{x}. \langle F, H' \wedge E, D \rangle \rightarrow \exists \bar{x}\bar{y}. \langle B \wedge F, H' \wedge E, H = H' \wedge D \rangle$
 if $(H \Rightarrow G \mid B)$ is in P renamed with fresh variables \bar{y} ,
 and $\mathcal{X} \models D \rightarrow \exists \bar{y}(H = H' \wedge G)$.

where the variables appearing in triples stand for conjunctions of constraints, and \bar{x} represents the set of variables appearing in the head H .

Example 4. One possible execution of the previous program is :

$$\begin{array}{ll}
 \langle Z = \langle X, X = \langle Y \wedge Y = \langle Z, true \rangle \rangle \rangle & \text{(Introduce } \times 2) \\
 \langle X = \langle Z \wedge Z = \langle X, X = \langle Y \wedge Y = \langle Z, true \rangle \rangle \rangle \rangle & \text{(Propagate transitivity)} \\
 \langle true, X = \langle Z \wedge Z = \langle X \wedge X = \langle Y \wedge Y = \langle Z, true \rangle \rangle \rangle \rangle & \text{(Introduce } \times 2) \\
 \langle X = Z, X = \langle Y \wedge Y = \langle Z, true \rangle \rangle \rangle & \text{(Simplify antisymmetry)} \\
 \langle true, X = \langle Y \wedge Y = \langle Z, X = Z \rangle \rangle \rangle & \text{(Solve)} \\
 \langle X = Y, true, X = Z \rangle & \text{(Simplify antisymmetry)} \\
 \langle true, true, X = Y \wedge X = Z \rangle & \text{(Solve)}
 \end{array}$$

Definition 13 (Quantified conjunction). *The quantified conjunction of two states is a binary operator $+_{\bar{y}}$, parameterized by a set \bar{y} of variables, defined by:*

$$\exists \bar{x}. \langle F, E, D \rangle +_{\bar{y}} \exists \bar{x}'. \langle F', E', D' \rangle = \exists \bar{y}\bar{x}\bar{x}'. \langle F \wedge F', E \wedge E', D \wedge D' \rangle$$

where \bar{y} , \bar{x} , \bar{x}' are supposed disjoint without loss of generality. Let \mathcal{C}_h be the family of quantified conjunction operators.

Proposition 12. *Quantified conjunctions are \rightarrow -linear.*

Unlike the orders defined in the previous section for first-order terms, the pre-order $\geq_{\mathcal{C}_h}$ may be not well-founded. This is the case when logical implication in \mathcal{X} is not well-founded. For example if \mathcal{X} is the constraint system (\mathbb{N}, \leq) , the chain $p_1 >_{\mathcal{C}_h} p_2 >_{\mathcal{C}_h} p_3 \dots$ (where $p_i = (\langle \emptyset, \emptyset, 1 \leq x \wedge x \leq i \rangle, \langle \emptyset, \emptyset, x \leq i \rangle)$) is strictly decreasing w.r.t. $>_{\mathcal{C}_h}$.

However one can prove that the brother pairs (\wedge) admit $\geq_{\mathcal{C}_h}$ -minimal elements. For this purpose, we assume without loss of generality that no rule in P is subsumed by another one in P , since P is finit. Here we will say that a simplification rule (resp. a propagation rule) R *subsumes* another rule $(H \Leftrightarrow G_1 \mid B)$ (resp. $(H', H \Rightarrow G_1 \mid B)$) if there exists a renaming of R of the form $(H \Leftrightarrow G_2 \mid B)$ (resp. $(H \Rightarrow G_2 \mid B)$) such that the constraint G_2 subsumes G_1 in \mathcal{X} .

Proposition 13. (\wedge) is lower bounded with respect to \geq_{c_h} .

Proof. Let min be the mapping of valid reductions to pairs of states defined as follows :

1. $min(\exists \bar{x}. \langle C \wedge F, E, D \rangle \rightarrow \exists \bar{x}. \langle F, E, C \wedge D \rangle) = (\langle C, \emptyset, \top \rangle, \langle \emptyset, \emptyset, C \rangle)$;
2. $min(\exists \bar{x}. \langle H \wedge F, E, D \rangle \rightarrow \exists \bar{x}. \langle F, H \wedge E, D \rangle) = (\langle H, \emptyset, \top \rangle, \langle \emptyset, H, \top \rangle)$;
3. $min(\exists \bar{x}. \langle F, H' \wedge E, D \rangle \rightarrow \exists \bar{x}\bar{y}. \langle B \wedge F, E, D \rangle) = (\langle \emptyset, H', G \rangle, \langle B, \emptyset, G \rangle)$
if $(H \Leftrightarrow G \mid B)$ is in P renamed with fresh variables \bar{y} ;
4. $min(\exists \bar{x}. \langle F, H' \wedge E, D \rangle \rightarrow \exists \bar{x}\bar{y}. \langle B \wedge F, H' \wedge E, D \rangle) = (\langle \emptyset, H', G \rangle, \langle B, H', G \rangle)$
if $(H \Rightarrow G \mid B)$ is in P renamed with fresh variables \bar{y}

By cases on the type of the reduction, one can check that min is a total function, i.e. any reduction is mapped to a pair. Moreover, for any reduction $S \rightarrow S'$, $min(S \rightarrow S') = (T, T')$ defines a reduction $T \rightarrow T'$ that is smaller than or equal to any other comparable reduction.

Now we prove that any pair (S_1, S_2) in (\wedge) is comparable to a minimal pair. Let S be a state such that $S \rightarrow S_1$ and $S \rightarrow S_2$. Let $(T_1, T'_1) = min(S \rightarrow S_1)$ and $(T_2, T'_2) = min(S \rightarrow S_2)$. We suppose that $T'_1 \neq T'_2$, otherwise (S_1, S_2) would be symmetrical. The proof is by case on T_1 and T_2 :

- T_1 (or T_2) is of the form $\langle H, \emptyset, \top \rangle$: since $T'_1 \neq T'_2$, H is not in the goal of T_2 (or T_1). Hence we deduce that $(S_1, S_2) \geq_{c_h} (T'_1 +_{\emptyset} T_2, T_1 +_{\emptyset} T'_2)$ that is clearly minimal in (\wedge) .
- T_1 (or T_2) is of the form $\langle C, \emptyset, \top \rangle$: as in the previous case we infer that $(S_1, S_2) \geq_{c_h} (T'_1 +_{\emptyset} T_2, T_1 +_{\emptyset} T'_2)$ that is minimal in (\wedge) .
- $T_1 = \langle \emptyset, H_1, C_1 \rangle$ and $T_2 = \langle \emptyset, H_2, C_2 \rangle$: let $T'_1 = \langle B_1, H'_1, C_1 \rangle$ and $T'_2 = \langle B_2, H'_2, C_2 \rangle$. Let $\{H_{11}, H_{12}\}$ and $\{H_{21}, H_{22}\}$ two partitions of H_1 and H_2 such that $H_{11} = H_{22}$ and $H_{12} \cap H_{21} = \emptyset$. Then we have $(S_1, S_2) \geq_{c_h} (\langle B_1, H'_1 \wedge H_{21}, G_1 \wedge G_2 \wedge H_{11} = H_{22} \rangle, \langle B_2, H'_2 \wedge H_{12}, G_1 \wedge G_2 \wedge H_{11} = H_{12} \rangle)$. The latter pair is minimal in (\wedge) , as otherwise H_{11}, H_{12} and H_{21}, H_{22} would not be the biggest partitions of H_1 and H_2 . \square

Let $\zeta(R)$ be the constraints in the head of R that are not deleted by its application, i.e. $\zeta(R) = true$ if R is a simplification or $\zeta(R) = H$ if R is a propagation rule with head H . Let R be a rule with guard G , body B and head H_1, \dots, H_n and let R' be a rule renamed with fresh variables with guard G' , body B' and head H'_1, \dots, H'_m . A *CHR critical pair* between R and R' is a pair of the form :

$$(\exists \bar{y}. \langle \zeta(R), H'_{j_{k+1}}, \dots, H_{i_m}, B, true, \bar{G} \rangle, \exists \bar{y}. \langle \zeta(R'), H_{i_{k+1}}, \dots, H_{i_n}, B', true, \bar{G} \rangle)$$

where $\bar{G} = G \wedge G' \wedge H_{i_1} = H'_{j_1} \wedge \dots \wedge H_{i_k} = H'_{j_k}$, while $\{i_1, \dots, i_n\}$ and $\{j_1, \dots, j_m\}$ are permutation of $\{1, \dots, n\}$ and $\{1, \dots, m\}$ respectively, and \bar{y} is the set of variables appearing in the bodies but not in the heads.

Proposition 14. Any C_h -critical pair is a *CHR critical pair*.

This shows the soundness of analyzing the confluence of CHR programs by computing CHR critical pairs [8]. However a CHR critical pair is not necessarily a \mathcal{C}_h critical pair, as shows the following :

Example 5. Let P be the CHR programe consisting into the two following rules $p, q, r \Leftarrow \text{case}(1)$ and $p, q, s \Leftarrow \text{case}(2)$. P admits three critical pairs :

$$\begin{aligned} p_1 &= (\langle \text{case}(1), s, \text{true} \rangle, \langle \text{case}(2), r, \text{true} \rangle) \\ p_2 &= (\langle \text{case}(1), q \wedge s, \text{true} \rangle, \langle \text{case}(2), q \wedge r, \text{true} \rangle) \\ p_3 &= (\langle \text{case}(1), p \wedge s, \text{true} \rangle, \langle \text{case}(2), p \wedge s, \text{true} \rangle) \end{aligned}$$

However, p_2 and p_3 are not \mathcal{C}_h -critical as $p_2 >_{\mathcal{C}_h} p_1$ and $p_3 >_{\mathcal{C}_h} p_1$.

5.3 CHR Under Its Refined Semantics

The refined operational semantics of CHR includes a partial control strategy that prevents the looping of propagation rules by restricting their firing only once on the same instances. By internalizing the necessary information in the states, one can nevertheless prove local confluence by computing critical pairs between states enriched with control tokens.

A *refined CHR state* $\exists \bar{x}. \langle \langle F, E, D, T \rangle \rangle$ is composed of a naive state $\exists \bar{x}. \langle F, E, D \rangle$ together with a set \mathcal{T} , the *token store*, composed of tokens of the form $\mathcal{R} @ C$ where C is a conjunction of constraints and \mathcal{R} a rule name. \mathcal{T} contains the necessary information about propagation rules, the respective constraints that can be possibly applied are contained in \mathcal{T} . Given a CHR program P and a CHR constraint C , the *tokenset* of an user-defined constraint C with respect to conjunction of constraints C_U is the set :

$$T(C, C_u) = \left\{ \mathcal{R} @ H' \mid \begin{array}{l} \mathcal{R} @ (H \implies G \mid B) \in P, C \text{ is in } H', \\ H' \text{ is a subset of } C \wedge C_u, H \text{ unifies with } H' \end{array} \right\}$$

and $T(C_1 \wedge \dots \wedge C_n, C_u) = T(C_1, C_u) \cup \dots \cup T(C_n, C_u)$. The refined transition relation \rightarrow_+ is given by the following rules, where the variables appearing in triples stand for conjunctions of constraints and \bar{x} represents the set of variables appearing in the head H .

Solve

$\exists \bar{x}. \langle \langle C \wedge F, E, D, T \rangle \rangle \rightarrow_+ \exists \bar{x}. \langle \langle F, E, C \wedge D, T \rangle \rangle$ if C is a built-in constraint

Introduce

$\exists \bar{x}. \langle \langle H \wedge F, E, D, T \rangle \rangle \rightarrow_+ \exists \bar{x}. \langle \langle F, H \wedge E, D, T \cup T(H, E) \rangle \rangle$
if H is a CHR constraint.

Simplify

$\exists \bar{x}. \langle \langle F, H' \wedge E, D, T \rangle \rangle \rightarrow_+ \exists \bar{x} \bar{y}. \langle \langle B \wedge F, E, D, T \cap T(H \wedge E, \top) \rangle \rangle$
if $(H \Leftarrow G \mid B)$ is in P renamed with fresh variables \bar{y} ,
and $\mathcal{X} \models D \rightarrow \exists \bar{x} (H = H' \wedge G)$.

Propagate

$\exists \bar{x}. \langle \langle F, H' \wedge E, D, \{\mathcal{R} @ H'\} \cup T \rangle \rangle \rightarrow_+ \exists \bar{x} \bar{y}. \langle \langle B \wedge F, H' \wedge E, D, T \rangle \rangle$
if $\mathcal{R} @ (H \implies G \mid B)$ is in P renamed with fresh variables \bar{y} ,
and $\mathcal{X} \models D \rightarrow \exists \bar{x} (H = H' \wedge G)$.

Definition 14 (Refined state conjunction). *The refined quantified conjunction of two states is a binary operator $+_{\bar{y}}$, parameterized by a set \bar{y} of variables and defined as:*

$$\exists \bar{x}. \langle\langle F, E, D, \mathcal{T} \rangle\rangle +_{\bar{y}} \exists \bar{x}'. \langle\langle F', E', D', \mathcal{T}' \rangle\rangle = \exists \bar{y} \bar{x} \bar{x}'. \langle\langle F \wedge F', E \wedge E', D \wedge D', \mathcal{T} \cup \mathcal{T}' \rangle\rangle$$

where \bar{y} , \bar{x} , \bar{x}' are supposed disjoint without loss of generality. Let \mathcal{C}_h^+ be the family of refined state conjunction operators.

Definition 15. *Let R be a rule with guard G , body B and head H_1, \dots, H_n and let R' be a rule renamed with fresh variables with guard G' , body B' and head H'_1, \dots, H'_m . A refined critical pair between R and R' is a pair of the form : $(\exists \bar{y}. \langle\langle \zeta(R), H'_{j_{k+1}}, \dots, H_{i_m} B, \text{true}, \tilde{G}, \emptyset \rangle\rangle, \exists \bar{y}. \langle\langle \zeta(R'), H_{i_{k+1}} \dots H_{i_n}, B', \text{true}, \tilde{G}, \emptyset \rangle\rangle)$ where $\tilde{G} = G \wedge G' \wedge H_{i_1} = H'_{j_1} \wedge \dots \wedge H_{i_k} = H'_{j_k}$, while $\{i_1, \dots, i_n\}$ and $\{j_1, \dots, j_m\}$ are permutation of $\{1, \dots, n\}$ and $\{1, \dots, m\}$ respectively and \bar{y} is the set of variables appearing in bodies but not in the heads.*

With a proof similar to the previous proposition, we get :

Proposition 15. *Any \mathcal{C}_h^+ -critical pair is a CHR refined critical pair.*

6 Conclusion

By abstracting the notion of critical pairs from term rewriting systems to arbitrary binary relations and arbitrary context operators over some set E , an abstract critical pair theorem has been proved, and shown useful to establish the local confluence of a wide variety of transition systems. This has been illustrated by instantiating the abstract notion of contexts and critical pairs to prove the soundness of classical critical pair definitions in term rewriting systems, conditional term rewriting systems, and Constraint Handling Rules programs. In the latter case, our use of bounded orderings instead of well-founded orderings has been shown necessary to handle the constrained states of CHR transitions. Interestingly in all these cases, we have shown that some classical critical pairs could be disregarded.

An abstract notion of linear contexts and linear decomposition has been proved useful to establish these results. This could be further developed to define an abstract notion of orthogonal systems [16]. As for future work, the generalization of our approach to n-ary relations might also be worth investigating in connection to the theory of canonical inference [3,6], with well-founded ordering assumptions replaced by boundedness conditions.

Acknowledgments. We are grateful to the anonymous referees for their comments and for pointing us to [3]. This work benefited from partial support by the ANR RNTL Manifico project.

References

1. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Smolka, G. (ed.) *Principles and Practice of Constraint Programming - CP97*. LNCS, vol. 1330, pp. 252–266. Springer, Heidelberg (1997)
2. Blanqui, F.: Termination and confluence of higher-order rewrite systems. In: *Proceedings of the 11th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, pp. 47–61 (2000)
3. Bonacina, M., Dershowitz, N.: Abstract canonical inference. *ACM Transactions on Computational Logic*, 8(1) (2007)
4. Clavel, M., Durán, F., Eker, S., Meseguer, J.: Building equational proving tools by reflection in rewriting logic. In: *Proceedings of the CafeOBJ Symposium '98*. Japan Advanced Institute for Science and Technology (1998)
5. Corradini, A., Montanari, U.: An algebra of graphs and graph rewriting. In: *Proceedings of the 4th International Conference on Category Theory and Computer Science*, Lecture Notes in Computer Science, pp. 236–260 (1991)
6. Dershowitz, N., Kirchner, C.: Abstract canonical presentations. *Journal of Theoretical Computer Science* 357, 53–69 (2006)
7. Dershowitz, N., Okada, M., Sivakumar, G.: Confluence of conditional rewrite systems. In: Kaplan, S., Jouannaud, J.-P. (eds.) *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*. LNCS, vol. 308, pp. 31–44. Springer, Heidelberg (1988)
8. Frühwirth, T.: Theory and practice of constraint handling rules. *Journal of Logic Programming*, Special Issue on Constraint Logic Programming 37(1-3), 95–138 (1998)
9. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM* 27(4), 797–821 (1980)
10. Jouannaud, J.-P., Kirchner, H.: Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing* 15(4), 1155–1194 (1986)
11. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: *Proceedings of the 11th International Conference on Concurrency Theory*, pp. 243–258 (2000)
12. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: a progress report. In: *Proceedings of the 7th International Conference on Concurrency Theory*, pp. 331–372 (1996)
13. Noll, T.: On coherence properties in term rewriting models of concurrency. In: Baeten, J.C.M., Mauw, S. (eds.) *CONCUR 1999*. LNCS, vol. 1664, pp. 478–493. Springer, Heidelberg (1999)
14. Peterson, G., Stickel, M.: Complete sets of reductions for some equational theories. *Journal of the ACM* 28(2), 233–264 (1981)
15. Raoult, J., Voisin, F.: Set-theoretic graph rewriting. In: *Proceedings of the International Workshop on Graph Transformations in Computer Science*, pp. 312–325 (1993)
16. Terese.: *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (2003)

On the Completeness of Context-Sensitive Order-Sorted Specifications*

Joe Hendrix and José Meseguer

University of Illinois at Urbana-Champaign
{jhendrix,meseguer}@uiuc.edu

Abstract. We propose three different notions of completeness for order-sorted equational specifications supporting context-sensitive rewriting modulo axioms relative to a replacement map μ . Our three notions are: (1) a definition of μ -canonical completeness under which μ -canonical forms coincide with canonical forms; (2) a definition of semantic completeness that guarantees that the μ -operational semantics and standard initial algebra semantics are isomorphic; and (3) an appropriate definition of μ -sufficient completeness with respect to a set of constructor symbols. Based on these notions, we use equational tree automata techniques to obtain decision procedures for checking these three kinds of completeness for equational specifications satisfying appropriate requirements such as weak normalization, ground confluence and sort-decreasingness, and left-linearity. The decision procedures are implemented as an extension of the Maude sufficient completeness checker.

1 Introduction

In equational programming there is a relentless drive to increase the expressiveness and generality of programs. This provides a much easier and elegant way of mapping many applications into such languages. For example, the use of sorts, subsorts, and matching modulo axioms like associativity and/or commutativity, makes equational programming much easier and allows very elegant and succinct programming solutions.

Another important dimension, along which program expressiveness can be substantially increased is that of user-programmable evaluation strategies based on *context-sensitive* (CS) rewriting (see, for example (14; 16; 22)). They allow very fine-grained control (at the level of each individual function symbol) on how the rewriting evaluation is performed. Their value and practical importance has been recognized in many equational languages. OBJ2 (6) was the first such language supporting them; and they are, for example, supported in all languages in the OBJ family, including CafeOBJ (5) and Maude (2). In practice, CS rewriting can be used for two somewhat different purposes:

1. to increase the *efficiency* of a standard equational program without changing its meaning: for example by restricting the evaluation of an if-then-else

* Research supported by ONR Grant N00014-02-1-0715.

symbol to its first, boolean argument to avoid wasteful or even nonterminating computations; and

2. as a way to compute with *infinite data structures* such as, for example, the infinite stream of all prime numbers, in a *lazy* way; in this second case, CS rewriting provides an elegant, finitary way of computing with infinite objects.

Expressiveness is substantially increased in both of these ways, since the user can *both* control the efficiency of program execution and map into the language new applications involving infinite data structures.

This is all very well. However, there are a number of open research questions about how to *reason* formally about equational programs supporting CS rewriting for verification purposes. Two areas where important progress has been made are in methods for proving termination, e.g., [7; 18; 22] and confluence [14] of CS equational programs. But other important questions remain unexplored.

Imagine, for example, that you want to use an inductive theorem prover to verify some property about a CS equational program. No inductive theorem prover that we are aware of allows reasoning about CS programs. Is it ok to *ignore* the CS information and just reason about the underlying equational theory? We think that, in general, the answer is: *definitely not!* Why not? Because inductive reasoning principles may not be sound on the *model* of the CS program.

What models are we talking about? Well, that is, indeed, one of the interesting research questions. For an equational *theory* (Σ, E) , the model on which inductive reasoning is sound is obvious, namely, the *initial algebra* $T_{\Sigma/E}$. In fact, *initial algebra semantics* is the standard mathematical semantics of equational programs in languages such as OBJ, CafeOBJ, and Maude. Furthermore, provided that the equational program is weakly normalizing and ground confluent, the initial algebra semantics fully *agrees* with the *operational semantics*, in the precise, mathematical sense that the initial algebra $T_{\Sigma/E}$ and the *canonical term algebra* $\text{Can}_{\Sigma/E}$ obtained by rewriting are *isomorphic*. For CS rewriting the matter is less obvious, since we only have an operational semantics provided by the CS rewriting relation, but as far as we know no mathematical models in the form of *algebras* have been put forward. Therefore, the first thing we do in this work is to put forward such an algebra, namely, the algebra $\text{Can}_{\Sigma/E}^{\mu}$ of *μ -canonical forms*, for μ the replacement map of the given CS program. We do so not just for vanilla-flavored, untyped CS programs, but for the more general and expressive CS programs with other features such as order-sorting and rewriting module axioms that one encounters in actual languages.

The importance of the algebra $\text{Can}_{\Sigma/E}^{\mu}$ is that it enables articulating and providing proof methods for three important CS *completeness problems*, namely:

1. *μ -canonical completeness*, which means satisfying the set-theoretic equality $\text{Can}_{\Sigma/E,s}^{\mu} = \text{Can}_{\Sigma/E,s}$ for each sort s in the specification;
2. *μ -semantic completeness*, which model-theoretically corresponds to the case where the surjective Σ -homomorphism $q : \text{Can}_{\Sigma/E}^{\mu} \rightarrow T_{\Sigma/E}$, which we show always exists under minimal assumptions, is an *isomorphism*, and

proof-theoretically means that the *sound* way of proving ground E -equalities by CS rewriting is also *complete*;

3. μ -sufficient completeness, which generalizes the usual sufficient completeness of equational function definitions with respect to a signature of constructors to the CS case. The subtlety here is that in general it would be *too strong* to require that constructors appear in all positions of a term t in μ -canonical form: we only make such a requirement for *replacing* positions in t .

Our goal is not only to articulate these notions, but also to provide proof methods for them in the form of *decision procedures* under mild assumptions about the given CS program. Given that the CS programs we consider perform rewriting modulo axioms and are order-sorted, our methods are based on *Propositional Tree Automata* (12), a kind of equational tree automata (20), that can take into account both sort information and reasoning modulo axioms. These decision procedures have been implemented in an extension of Maude’s Sufficient Completeness Checker (SCC) (11), and we use several Maude programs to illustrate both the basic ideas and the use of SCC in verifying CS completeness properties.

The paper is organized as follows. In Section 2, we review basic concepts from order-sorted algebra, and introduce the precise class of CS term-rewrite systems we are considering. In Section 3, we define the canonical term algebra for a CS specification. In Section 4, we define the three notions of CS completeness, and in Section 5 we show how one can use PTA to check these completeness notions under appropriate assumptions. Finally, we discuss related work and suggest future avenues of research in Section 6. Full proofs can be found in (10).

2 Preliminaries

2.1 Order-Sorted Algebra

Order-sorted algebras are an extension of many-sorted algebras where a partial order \leq is associated to the sorts in order to build a notion of subtype and supertype into the algebra and operators can be overloaded and then must agree on common data.

Definition 1. An order-sorted signature is a tuple $\Sigma = (S, F, \leq)$ where

- (S, \leq) is a poset; and
- $F = \{F_{w,s}\}_{(w,s) \in S^* \times S}$ is a family of operator symbols such that if $f \in F_{w,s} \cap F_{w',s'}$ then $w \equiv_{\leq} w'$ and $s \equiv_{\leq} s'$, where \equiv_{\leq} denotes the equivalence relation generated by \leq extended to sequences in the usual way.

We assume the existence of an S -sorted family of variables $X = \{X_s\}_{s \in S}$ distinct from the operators F , where each set X_s is countably infinite and pairwise disjoint. We write x_s if x is a variable in X_s . When the signature $\Sigma = (S, F, \leq)$ is clear, we sometimes write $f : s_1 \dots s_n \rightarrow s$ for $f \in F_{s_1 \dots s_n, s}$. Given a signature Σ , $T_{\Sigma}(X)_s$ denotes the terms with sort s formed by the operators in Σ and

variables in X , and $T_{\Sigma,s}$ denotes the ground terms with sort s . A *substitution* $\theta : X \rightarrow T_{\Sigma}(X)$ is a mapping such that $\theta(x_s) \in T_{\Sigma}(X)_s$ for each $x_s \in X_s$,

Definition 2. An order-sorted theory is a pair $\mathcal{E} = (\Sigma, E)$ where $\Sigma = (S, F, \leq)$ is an order-sorted signature, and E is a set of equations of the form $l = r$ with $l, r \in T_{\Sigma}(X)$ terms having sorts in the same equivalence class in S / \equiv_{\leq} .

There are various inference systems in order-sorted logic for deriving equations of the form $t = u$. We use the sound and complete inference system presented in (19). We also use the definition for order-sorted algebras and homomorphisms found in (19). See (9; 19) for surveys on order-sorted algebra.

Definition 3. A Σ -algebra A for a signature $\Sigma = (S, F, \leq)$ consists of:

- a set A_s for each sort $s \in S$ such that $A_s \subseteq A_{s'}$ for $s \leq s'$;
- a function $A_{f:w \rightarrow s} : A_w \rightarrow A_s$ for each symbol $f \in F_{w,s}$ where $w = s_1 \dots s_n$, $A_w = A_{s_1} \times \dots \times A_{s_n}$, and $A_{f:w \rightarrow s}(\bar{a}) = A_{f:w' \rightarrow s'}(\bar{a})$ for each $f \in F_{w,s} \cap F_{w',s'}$ and $\bar{a} \in A_w \cap A_{w'}$.

A Σ -homomorphism $h : A \rightarrow B$ is a family of functions $\{h_s : A_s \rightarrow B_s\}_{s \in S}$ such that: if $s \equiv_{\leq} s'$ and $a \in A_s \cap A_{s'}$, then $h_s(a) = h_{s'}(a)$; and for $f \in F_{s_1 \dots s_n, s}$ and $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$, we have $h_s(A_f(a_1, \dots, a_n)) = B_f(h_{s_1}(a_1), \dots, h_{s_n}(a_n))$.

Given an order-sorted theory $\mathcal{E} = (\Sigma, E)$, an \mathcal{E} -algebra is a Σ -algebra satisfying the equations in E . We let T_{Σ} denote the term algebra for Σ , and $T_{\Sigma/E}$ denote the \mathcal{E} -algebra such that $T_{\Sigma/E,s} = \{[t]_E \mid t \in T_{\Sigma,s}\}$ for each sort $s \in S$, where $[t]_E$ denotes the equivalence class of t under $=_E$. Both T_{Σ} and $T_{\Sigma/E}$ are *initial* for the categories of Σ -algebras and \mathcal{E} -algebras respectively, so there is a unique homomorphism from T_{Σ} to any Σ -algebra, and a unique homomorphism from $T_{\Sigma/E}$ to any \mathcal{E} -algebra. For a Σ -algebra A and term $t \in T_{\Sigma}$, we let $A(t)$ denote the value of t in the unique homomorphism $A : T_{\Sigma} \rightarrow A$, i.e., $A(f(t_1, \dots, t_n)) = A_f(A(t_1), \dots, A(t_n))$.

2.2 Context-Sensitive Order-Sorted Term Rewrite Systems

In order to execute an equational theory, one typically treats the equations $l = r \in E$ as rewrite rules $l \rightarrow r$ and simplifies expressions from left to right. The most advanced rewrite engines today have matching algorithms capable of matching modulo specific equational axioms such as associativity and commutativity, and with such systems we treat those specific axioms as equations while treating the other axioms as rules. When rewriting modulo axioms, the variables in the axioms are typically constrained at the level of the connected component rather than of individual sorts. We include this restriction in the definition below:

Definition 4. An Order-sorted Term Rewrite System (TRS) is a tuple $\mathcal{R} = (\Sigma, A, R)$ where:

- $\Sigma = (S, F, \leq)$ is an order-sorted signature where each connected component $[s] \in S / \equiv_{\leq}$ contains a maximal sort denoted by k_s ;

- A is a set of unconditional Σ -equations where the variables in each equation are only constrained with the maximal sorts; and
- R is a set of rewrite rules of the form $l \rightarrow r$ with $l, r \in T_\Sigma(X)_s$ for some $s \in S$, and $\text{vars}(r) \subseteq \text{vars}(l)$;

Given a TRS $\mathcal{R} = (\Sigma, A, R)$, $\text{lhs}(R)$ denotes the left-hand sides of the rules in R , i.e., $\text{lhs}(R) = \{l \mid l \rightarrow r \in R\}$. We say that an order-sorted TRS $\mathcal{R} = (\Sigma, A, R)$ is *left-linear* if each $l \in \text{lhs}(R)$ is linear. Due to the restrictions on the signatures and equations in our term rewrite system we are able to treat our order-sorted axioms as many-sorted axioms for the purposes of matching modulo.

Definition 5. Given an order-sorted TRS $\mathcal{R} = (\Sigma, A, R)$ with $\Sigma = (S, F, \leq)$, we let $\Sigma^k = (S^K, F^K)$ denote the many-sorted signature where S^K contains exactly the maximal sorts $k_s \in S$, and F^K contains an operator $f : k_{s_1} \dots k_{s_n} \rightarrow k_s$ for each $f \in F_{s_1 \dots s_n, s}$.

Due to our restrictions on the equations in our term-rewrite theories, we can essentially use $=_A$ to denote $=_{(\Sigma, A)}$ and $=_{(\Sigma^k, A)}$ interchangeably on ground terms, as justified by the following lemma which can be easily proved:

Lemma 1. Given a order-sorted TRS $\mathcal{R} = (\Sigma, A, R)$, for all terms $t, u \in T_\Sigma$, we have $t =_{(\Sigma, A)} u$ iff $t =_{(\Sigma^k, A)} u$.

We are interested in studying and analyzing CS rewriting for order-sorted term rewrite systems. In CS rewriting, there is a function $\mu : F \rightarrow \mathcal{P}(\mathbb{N})$, called the *replacement map*, which maps each function symbol $f \in F$ to a set of *replacing positions* $\mu(f) \subseteq \{1, \dots, \text{arity}(f)\}$. The replacement map μ is used for restricting rewriting so that in rewriting a term $f(t_1, \dots, t_n) \in T_\Sigma(X)$, the term t_i can only be rewritten if $i \in \mu(f)$. A *CS term rewrite system* is a pair (\mathcal{R}, μ) where μ is a replacement map for the signature used in \mathcal{R} .

Given a replacement map μ , the set of positions that may be rewritten are called the μ -*replacing positions* and denoted by $\text{pos}^\mu(t)$. Formally, we have:

$$\text{pos}^\mu(x) = \{\epsilon\} \quad \text{and} \quad \text{pos}^\mu(f(t_1, \dots, t_n)) = \{\epsilon\} \cup \bigcup_{i \in \mu(f)} \{iw \mid w \in \text{pos}^\mu(t_i)\}.$$

A context C is μ -*replacing* when the hole appears in a μ -replacing position.

We write $t \rightarrow_{\mathcal{R}, \mu} u$ if t rewrites to u using the rules in \mathcal{R} and replacement map μ in a *single* rewrite step, i.e., there is a rule $l \rightarrow r$ in R such that $t =_A C[l\theta]$ and $u =_A C[r\theta]$ for some μ -replacing context C and substitution $\theta : X \rightarrow T_\Sigma(X)$. The reflexive and transitive of closure of $\rightarrow_{\mathcal{R}, \mu}$ is $\rightarrow_{\mathcal{R}, \mu}^*$. We write $t \downarrow_{\mathcal{R}}^\mu u$ if t and u can be rewritten to the same term, i.e., there is a term $v \in T_\Sigma(X)$ such that $t \rightarrow_{\mathcal{R}, \mu}^* v$ and $u \rightarrow_{\mathcal{R}, \mu}^* v$.

A term $t \in T_\Sigma(X)$ is (\mathcal{R}, μ) -*reducible* iff there is a $u \in T_\Sigma(X)$ such that $t \rightarrow_{\mathcal{R}, \mu} u$, and (\mathcal{R}, μ) -*irreducible* otherwise. We also say that a μ -irreducible term $t \in T_\Sigma(X)$ is in μ -*canonical* form. We write $t \rightarrow_{\mathcal{R}, \mu}^! u$ if $t \rightarrow_{\mathcal{R}, \mu}^* u$ and u is (\mathcal{R}, μ) -irreducible. \mathcal{R} is μ -*weakly normalizing* when for each term $t \in T_\Sigma(X)$ there

is a term $u \in T_\Sigma(X)$ such that $t \rightarrow_{\mathcal{R},\mu}^! u$. \mathcal{R} is μ -terminating if the relation $\rightarrow_{\mathcal{R},\mu}$ is Noetherian. \mathcal{R} is μ -confluent if for all $t, u, v \in T_\Sigma(X)$, $t \rightarrow_{\mathcal{R},\mu}^* u$ and $t \rightarrow_{\mathcal{R},\mu}^* v$ implies $u \downarrow_{\mathcal{R}}^\mu v$. \mathcal{R} is μ -sort-decreasing if for all terms $t \in T_\Sigma(X)_s$ and $u \in T_\Sigma(X)_{k_s}$, $t \rightarrow_{\mathcal{R},\mu}^* u$ implies that there is a term $v \in T_\Sigma(X)_s$ such that $u \rightarrow_{\mathcal{R},\mu}^* v$. When \mathcal{R} is μ -weakly normalizing, μ -confluent, or μ -sort-decreasing on ground terms, we say that it is ground μ -weakly normalizing, ground μ -confluent, or ground μ -sort-decreasing, respectively.

When the replacement map μ allows rewriting at every subterm position, this inference system specializes to rewriting in the ordinary sense. Let μ_\top be the replacement map $f \mapsto \{1, \dots, \text{arity}(f)\}$. We write $t \rightarrow_{\mathcal{R}} u$ iff $t \rightarrow_{\mathcal{R},\mu_\top} u$, and $t \rightarrow_{\mathcal{R}}^* u$ iff $t \rightarrow_{\mathcal{R},\mu_\top}^* u$. Additionally, we will say that a system \mathcal{R} is *weakly normalizing* iff it is μ_\top -weakly normalizing. More generally, we extend this convention to all other properties. For example, we say that \mathcal{R} is confluent iff it is μ_\top -confluent.

3 Context-Sensitive Canonical Term Algebras

When $\mathcal{R} = (\Sigma, A, R)$ is ground μ -weakly normalizing and ground μ -confluent, for each term $t \in T_\Sigma$, there is a (\mathcal{R}, μ) -irreducible term, denoted by $t!_{\mathcal{R}}^\mu$ such that $t \rightarrow_{\mathcal{R},\mu}^! t!_{\mathcal{R}}^\mu$, which is unique up to A . When \mathcal{R} is additionally sort-decreasing, we can then define a (Σ, A) -algebra of (\mathcal{R}, μ) -canonical forms as follows:

Definition 6. Let $\mathcal{R} = (\Sigma, A, R)$ be a TRS with $\Sigma = (S, F, \leq)$ that is ground μ -weakly normalizing, ground μ -confluent and ground μ -sort-decreasing. The canonical term algebra for (\mathcal{R}, μ) is the Σ -algebra $\text{Can}_{\mathcal{R}}^\mu$ such that:

- for each sort $s \in S$, $\text{Can}_{\mathcal{R},s}^\mu = \{ [t]_A \in T_{\Sigma/A,s} \mid t \text{ is } (\mathcal{R}, \mu)\text{-irreducible} \}$; and
- for each $f \in F_{w,s}$, $\text{Can}_{\mathcal{R},f:w \rightarrow s}^\mu([t_1]_A, \dots, [t_n]_A) = [f(u_1, \dots, u_n)!_{\mathcal{R}}^\mu]_A$ where $u_i \in [t_i]_A \cap T_{\Sigma,s_i}$ for $1 \leq i \leq n$.

The algebra $\text{Can}_{\mathcal{R}}^\mu$ has a strong computation meaning: it is exactly the algebra of *values* (μ -normal forms) that a user interacting with a system that evaluates (\mathcal{R}, μ) obtains by reduction.¹ Therefore, it provides the perfect algebra for the *operational semantics* of (\mathcal{R}, μ) . This model is in a sense situated *between* the initial algebra T_Σ , and the model for the *mathematical semantics* of \mathcal{R} as an equational theory, namely, the initial algebra $T_{\Sigma/A \cup R}$. On the one hand, by initiality we have a unique homomorphism $\text{Can}_{\mathcal{R}}^\mu : T_\Sigma \rightarrow \text{Can}_{\mathcal{R}}^\mu$ which, as shown below, may not be surjective. On the other hand, $\text{Can}_{\mathcal{R}}^\mu$ is *more concrete* than $T_{\Sigma/A \cup R}$, and therefore a *sound*, but not necessarily complete, model for equational computation with \mathcal{R} . That is, we have:

Proposition 1. If $\mathcal{R} = (\Sigma, A, R)$ with $\Sigma = (S, F, \leq)$ is ground μ -weakly normalizing, ground μ -confluent, and ground μ -sort-decreasing, then the family of functions $\{q_s : \text{Can}_{\mathcal{R},s}^\mu \rightarrow T_{\Sigma/A \cup R,s}\}_{s \in S}$ with $q_s : [t]_A \mapsto [t]_{A \cup R}$ defines a surjective Σ -homomorphism $q : \text{Can}_{\mathcal{R}}^\mu \rightarrow T_{\Sigma/A \cup R}$.

¹ If (\mathcal{R}, μ) is μ -terminating, this is exactly true; if it is only μ -weakly normalizing, this requires either a μ -normalizing strategy, or the use of breadth-first search.

One typically constructs ground terminating and confluent specifications in order to reason about the equivalence of two terms algebraically, and it is important to be able to reduce the equality problem $t =_{AUR} u$ to the convergence problem $t \downarrow_{\mathcal{R}}^{\mu} u$. When considering ordinary (not context-sensitive) rewriting, we have $t =_{AUR} u$ iff $t \downarrow_{\mathcal{R}} u$ iff $t !_{\mathcal{R}} =_A u !_{\mathcal{R}}$ for terms $t, u \in T_{\Sigma}$ when \mathcal{R} is ground weakly normalizing, ground confluent, and ground sort-decreasing. In this case, we are guaranteed that $\text{Can}_{\mathcal{R}} = \text{Can}_{\mathcal{R}}^{\mu \top}$ is isomorphic to $T_{\Sigma/AUR}$, thus obtaining a perfect agreement between the operational semantics of \mathcal{R} and the mathematical, initial algebra semantics. In general, as we show below, this is *not* the case for CS rewriting, even if \mathcal{R} is ground μ -terminating, ground μ -confluent, and ground μ -sort-decreasing. That is $\text{Can}_{\mathcal{R}}^{\mu}$ is *sound*, since $t \downarrow_{\mathcal{R}}^{\mu} u$ implies $t =_{AUR} u$, but in general is *not complete*, i.e., $t =_{AUR} u \not\Rightarrow t \downarrow_{\mathcal{R}}^{\mu} u$.

Consider the specification \mathcal{R} with single sort s , symbols $a : \rightarrow s, b : \rightarrow s$, and $f : s \rightarrow s$, and replacement map μ where $\mu(f) = \emptyset$ with the rules: $a \rightarrow f(a)$ and $b \rightarrow f(a)$. This specification is clearly μ -weakly normalizing, μ -confluent, and μ -sort-decreasing. However, $T_{\Sigma/AUR,s} = \{[a]_{AUR}\}$ whereas $\text{Can}_{\mathcal{R},s}^{\mu}$ is the infinite set $\{\{f(a)\}, \{f(b)\}, \{f(f(a))\}, \{f(f(b))\}, \dots\}$.

The algebra $\text{Can}_{\mathcal{R}}^{\mu}$ differs from $\text{Can}_{\mathcal{R}}$ in several other properties as well. In general, it is not the case that $\text{Can}_{\mathcal{R}}^{\mu}(t) = t !_{\mathcal{R}}^{\mu}$. In the specification above, $\text{Can}_{\mathcal{R}}^{\mu}(f(a)) = f(\text{Can}_{\mathcal{R}}^{\mu}(a)) !_{\mathcal{R}}^{\mu} = f(f(a))$, whereas $f(a) !_{\mathcal{R}}^{\mu} = f(a)$. Additionally, the unique homomorphism $\text{Can}_{\mathcal{R}}^{\mu} : T_{\Sigma/A} \rightarrow \text{Can}_{\mathcal{R}}^{\mu}$ is neither surjective nor idempotent. For example, there is no term $t \in T_{\Sigma}$, such that $\text{Can}_{\mathcal{R}}^{\mu}(\{t\}) = \{f(b)\}$, while $\text{Can}_{\mathcal{R}}^{\mu}(\{a\}) = \{f(a)\}$ and $\text{Can}_{\mathcal{R}}^{\mu}(\{f(a)\}) = \{f(f(a))\}$.

4 Completeness in Context-Sensitive Rewriting

We have now shown that the usual requirements of μ -termination, μ -confluence, and μ -sort-decreasingness are insufficient to guarantee that the operational semantics of CS term rewriting corresponds to the mathematical semantics of the equational specification. One of the goals of this section is investigating what additional conditions we need to impose to guarantee that CS rewriting serves as a sound and complete technique to deduce ground equalities, i.e., when is the canonical term algebra $\text{Can}_{\mathcal{R}}^{\mu}$ isomorphic to the initial algebra $T_{\Sigma/AUR}$.

In this section, we introduce three notions of completeness for CS term rewrite systems (\mathcal{R}, μ) : (1) μ -canonical completeness; (2) μ -semantic completeness; and (3) μ -sufficient completeness. The first two notions of completeness are used to characterize the deductive power of CS rewriting. The third is used to analyze specifications that may not be complete in the first two senses, but may nevertheless represent useful applications of CS rewriting, such as specifying infinite data-structures. Later, in Section 5, we will show how these three completeness properties can be checked for specifications satisfying appropriate requirements such as left-linearity, ground μ -weak normalization, ground μ -confluence, and ground μ -sort-decreasingness.

4.1 μ -Canonical Completeness

The first property we consider is whether the canonical forms of CS rewriting and ordinary rewriting agree:

Definition 7. A TRS $\mathcal{R} = (\Sigma, A, R)$ is μ -canonically complete if every (\mathcal{R}, μ) -irreducible term $t \in T_\Sigma$ is \mathcal{R} -irreducible.

The theorem below shows that, for specifications that are ground μ -weakly normalizing and ground confluent, canonical completeness is enough to imply that CS and ordinary rewriting agree on convergence relations.

Theorem 1. If a TRS \mathcal{R} is ground μ -weakly normalizing, μ -canonically complete, and ground confluent, then for $t, u \in T_\Sigma$, $t \downarrow_{\mathcal{R}} u$ iff $t \downarrow_{\mathcal{R}}^\mu u$.

As a corollary, we observe that this class of specifications is μ -confluent.

Corollary 1. If \mathcal{R} is ground μ -weakly normalizing, μ -canonically complete, and ground confluent, then \mathcal{R} is ground μ -confluent.

In a similar vein, we can show ground μ -sort-decreasingness of \mathcal{R} by showing that \mathcal{R} is ground μ -weakly normalizing, μ -canonically complete, and sort-decreasing.

Theorem 2. If \mathcal{R} is ground μ -weakly normalizing, μ -canonically complete, and ground sort-decreasing, then \mathcal{R} is ground μ -sort-decreasing.

Together, Corollary 1 and Theorem 2 provide a means to check μ -confluence and μ -sort-decreasingness for μ -weakly normalizing, μ -canonically complete, confluent, and sort-decreasing specifications. Since one can prove μ -termination with existing tools (4; 8; 17), and check μ -canonical completeness of left-linear specifications with the decision procedure in Section 5.2, this eliminates the need for specialized CS-aware checking procedures for this class of specifications. The case of ground weak normalization and ground μ -weak normalization for μ -canonically complete specification yields a relation in the other direction.

Theorem 3. If \mathcal{R} is ground μ -weakly normalizing and μ -canonically complete, then \mathcal{R} is ground weakly normalizing.

On the other hand, if \mathcal{R} is ground weakly normalizing and μ -canonically complete, it may not be ground μ -weakly normalizing. Let \mathcal{R} have the rules: $f(x) \rightarrow f(x)$, $a \rightarrow b$, and $f(b) \rightarrow b$. \mathcal{R} is ground weakly normalizing, because every term can reduce to the \mathcal{R} -irreducible term b . Given the replacement map μ with $\mu(f) = \emptyset$, \mathcal{R} is μ -canonically complete, because b is the only (\mathcal{R}, μ) -irreducible term as well. However, \mathcal{R} is not μ -weakly normalizing, because $f(a) \not\rightarrow_{\mathcal{R}, \mu}^* b$.

As an example of a μ -canonically complete specification, we present a Maude module below for computing the factorial of a natural number.

```

fmod FACTORIAL is protecting NAT .
  var X Y Z : Nat .
  op p : Nat -> Nat .
  eq p(s(X)) = X .   eq p(0) = 0 .
  op if0 : Nat Nat Nat -> Nat [strat(1 0)].
  eq if0(0, Y, Z) = Y .   eq if0(s(X), Y, Z) = Z .
  op fact : Nat -> Nat .
  eq fact(X) = if0(X, s(0), X * fact(p(X))) .
endfm

```

This specification protects the built-in NAT specification, which contains constructor operators 0 and s for for zero and successor respectively, along with defined operators for plus and times. Predecessor p is defined as usual, and the operator if0 is annotated with a strategy $\text{strat}(1\ 0)$, indicating that only the first argument should be evaluated. Since the other operators are not given a strategy, Maude uses its default strategy, which evaluates every argument. In effect, these declarations define a replacement map μ where $\mu(\text{if0}) = \{1\}$ and $\mu(f) = \{1, \dots, \text{arity}(f)\}$ for $f \neq \text{if0}$. Using if0 , factorial can be defined with a single equation.

Without the strategy declaration on if0 , this specification is not terminating, and evaluating $\text{fact}(0)$ quickly leads to a segmentation fault in the Maude interpreter. However, with the given replacement map μ , the specification is μ -terminating. Moreover, it is μ -canonically complete, ground μ -confluent, and ground μ -sort-decreasing. Since there is only one sort, μ -sort-decreasingness is obvious. As the specification is left-linear, the decision procedure we introduce in Section 5.2 will allow us to automatically check its μ -canonically completeness. To see that it is ground μ -confluent one can just observe that it is confluent (indeed, orthogonal), and use Corollary 4.

4.2 μ -Semantic Completeness

Canonical completeness means that $\text{Can}_{\mathcal{R},s} = \text{Can}_{\mathcal{R},s}^{\mu}$ for each sort $s \in S$. By itself, this is not enough to immediately imply that $\text{Can}_{\mathcal{R}}^{\mu}$ and $T_{\Sigma/A \cup R}$ are isomorphic. This is implied by another notion of completeness, called μ -semantic completeness, which we define below.

Definition 8. A TRS $\mathcal{R} = (\Sigma, A, R)$ is μ -semantically complete iff for all $t, u \in T_{\Sigma}$, $t \downarrow_{\mathcal{R}}^{\mu} u$ iff $t =_{A \cup R} u$.

This definition at the syntactic level of terms captures the agreement between operational semantics and mathematical semantics that we want when the canonical algebra $\text{Can}_{\mathcal{R}}^{\mu}$ is well-defined.

Theorem 4. If \mathcal{R} is ground μ -weakly normalizing, ground μ -confluent, and ground μ -sort-decreasing, then \mathcal{R} is μ -semantically complete iff $\text{Can}_{\mathcal{R}}^{\mu}$ is isomorphic to $T_{\Sigma/A \cup R}$.

The next question that we address is how to check that a specification is μ -semantically complete. The results in the previous section on μ -canonical completeness lead to the following result:

Theorem 5. *A TRS \mathcal{R} that is ground μ -weakly normalizing, μ -canonically complete, ground confluent, and ground sort-decreasing is μ -semantically complete.*

As a corollary, we can easily obtain checkable conditions under which all three of the algebras $\text{Can}_{\mathcal{R}}^{\mu}$, $\text{Can}_{\mathcal{R}}$ and $T_{\Sigma/AUR}$ are isomorphic.

Corollary 2. *If \mathcal{R} is ground μ -weakly normalizing, ground μ -canonically complete, ground confluent, and ground sort-decreasing, then $\text{Can}_{\mathcal{R}}^{\mu}$ and $\text{Can}_{\mathcal{R}}$ are both well-defined and isomorphic to $T_{\Sigma/AUR}$.*

When the specification \mathcal{R} is ground μ -weakly normalizing, ground confluent, and ground sort-decreasing, μ -canonical completeness is a *sufficient* condition to show μ -semantic completeness, but it turns out not to be a *necessary* condition. For example, let \mathcal{R} have the rules: $f(f(x)) \rightarrow f(x)$, $a \rightarrow b$, and $f(b) \rightarrow f(a)$, and let μ be the replacement map with $\mu(f) = \emptyset$. The initial algebra contains two equivalence classes: one with the constants a and b , the other with terms containing f . The (\mathcal{R}, μ) -canonical terms are b and $f(a)$, and it is easy to show that $\text{Can}_{\mathcal{R}}^{\mu}$ and $T_{\Sigma/AUR}$ are isomorphic. Since \mathcal{R} is also ground μ -weakly normalizing, ground μ -confluent and ground μ -sort decreasing, \mathcal{R} is μ -semantically complete by Theorem 4. However, $f(a)$ is \mathcal{R} -reducible, leaving b the only \mathcal{R} -irreducible term, and so \mathcal{R} is not μ -canonically complete. In addition to not being μ -canonically complete, \mathcal{R} is not ground weakly normalizing. However, if \mathcal{R} is ground weakly normalizing, μ -semantic completeness implies μ -canonical completeness.

Theorem 6. *If \mathcal{R} is ground weakly normalizing and μ -semantically complete, then \mathcal{R} is μ -canonically complete.*

In other words, if \mathcal{R} is ground weakly normalizing and not μ -canonically complete, it is not μ -semantically complete either.

4.3 μ -Sufficient Completeness

Although μ -canonical completeness and μ -semantic completeness are useful notions of completeness in CS rewriting, there are many interesting applications of CS rewriting, especially those involving infinite data structures, that are not μ -semantically complete. As an example, we present a typed version of a specification of infinite lists from (16) in Maude syntax:

```
fmod INF-LIST is protecting NAT .
  sorts Nat? List .      subsort Nat < Nat? .
  op none : -> Nat? [ctor].
  op [] : -> List [ctor].
  op _:_ : Nat List -> List [ctor strat(1 0)].
  vars M N : Nat . var L : List .
  op sel : Nat List -> Nat? .
  eq sel(0, N : L) = N . eq sel(s(M), N : L) = sel(M, L) .
```

```

op from : Nat -> List .
eq from(M) = M : from(s(M)) .
op first : Nat List -> List .
eq first(0, L) = [] .   eq first(s(M), N : L) = N : first(M, L) .
endfm

```

The term `from(M)` represents the infinite list “ $M : M + 1 : \dots$ ”, and there are functions for obtaining the i th element in a list and the first n elements in the list. This specification is an interesting use of CS rewriting to obtain a terminating method to execute a non-terminating rewrite system. Although the equation for `from` is non-terminating, it is μ -terminating because of the strategy on “:”.

The specification `INF-LIST` is not μ -canonically complete, and its canonical algebra is not isomorphic to the initial algebra of the equational theory given by its axioms. For example $0 : \text{from}(s(0))$ and $0 : s(0) : \text{from}(s(s(0)))$ are distinct μ -canonical terms, but $0 : \text{from}(s(0)) =_{\text{INF-LIST}} 0 : s(0) : \text{from}(s(s(0)))$. In order to check properties of specifications like `INF-LIST` that are not μ -semantically complete, we therefore need techniques that analyze CS specifications directly. The case of μ -termination is well understood [7; 18; 22], and the case of μ -confluence has already been studied in [14].

Another interesting property that seems not to have been studied for CS specifications is sufficient completeness. Sufficient completeness in term-rewriting specifications means that enough equations have been defined so that all terms reduce to constructor terms. For example, a sufficiently complete specification involving arithmetic over the natural numbers should reduce every term containing plus and times to a term containing only zero and successor.

Although simple, this definition of sufficient completeness seems *too strong* in the context of CS specifications. The reason is that the non-replacing positions of a symbol intentionally do not reduce their arguments. Accordingly, our definition of μ -sufficient completeness allows defined symbols in the non-replacing positions of canonical terms, provided that all *replacing* positions have constructor symbols.

Definition 9. Let \mathcal{R} be a ground μ -weakly normalizing and ground μ -sort-decreasing TRS $\mathcal{R} = (\Sigma, A, R)$ with $\Sigma = (S, F, \leq)$ equipped with a indexed family of constructor symbols $C = \{C_{w,s}\}_{(w,s) \in S^* \times S}$ with each $C_{w,s} \subseteq F_{w,s}$. We say that \mathcal{R} is μ -sufficiently complete relative to C iff for all (\mathcal{R}, μ) -irreducible terms $t \in T_\Sigma$, $\text{pos}^\mu(t) \subseteq \text{pos}_C(t)$ where

$$\text{pos}_C(t) = \{i \in \text{pos}(t) \mid t_i = c(\bar{t}) \wedge c \in C_{w,s} \wedge \bar{t} \in T_{\Sigma,w}\}.$$

Our definition of μ -sufficient completeness reduces to the usual definition of sufficient completeness when every position is a replacing position, i.e., $\mu = \mu_\top$. Therefore, we say in this paper that a specification \mathcal{R} is *sufficiently complete* relative to C iff it is μ_\top -sufficiently complete relative to C .

Theorem 7. If \mathcal{R} is ground μ -weakly normalizing, μ -canonically complete, and ground sort-decreasing, then \mathcal{R} is μ -sufficiently complete relative to C iff it is sufficiently complete relative to C .

5 Checking μ -Completeness Properties

In the left-linear case, we are able to reduce the μ -canonical completeness and μ -sufficient completeness properties to an emptiness problem for *Propositional Tree Automata*, a class of tree automata introduced in (12) which is closed under Boolean operations and an equational theory. We are further able to use the results of Theorem 5 to have sufficient conditions for showing the μ -semantic completeness of \mathcal{R} when \mathcal{R} is left-linear, μ -weakly normalizing, μ -canonically complete, ground confluent, and ground sort-decreasing.

5.1 Propositional Tree Automata

We now define *Propositional Tree Automata*, first introduced in (12). We extend the definition of (12) from unsorted signatures to many-sorted signatures. We also use production rules $\alpha := f(\beta_1, \dots, \beta_n)$ in lieu of rewrite rules $f(\beta_1, \dots, \beta_n) \rightarrow \alpha$ in the definition to reflect a change in how the rules are interpreted. The definition using rewrite rules in (12) and the definition below are equivalent when the equations in the signature are linear. They are not equivalent, in general, when considering non-linear equations such as idempotence $f(x, x) = x$. We think that the definition given below is more useful in applications involving non-linear equations, and if we did not use this formalization, we would have to restrict later results in this paper involving tree automata to the linear case.

Definition 10. A Propositional Tree Automaton (PTA) $\mathcal{A} = (\mathcal{E}, Q, \Phi, \Delta)$ is a tuple in which:

- $\mathcal{E} = (\Sigma, E)$ is an many-sorted equational theory with $\Sigma = (S, F)$;
- $Q = \{Q_s\}_{s \in S}$ is a S -indexed family of sets of states disjoint from the function symbols in F ;
- $\Phi = \{\phi_s\}_{s \in S}$ is a S -indexed family of propositional formulae where the atomic propositions in ϕ_s are states in Q_s ; and
- Δ contains transition rules, each with one of the following forms: (1) $\alpha := f(\beta_1, \dots, \beta_n)$ where $f \in F_{s_1 \dots s_n, s}$, $\alpha \in Q_s$, and each $\beta_i \in Q_{s_i}$ for $1 \leq i \leq n$; or (2) $\alpha := \beta$ where $\alpha, \beta \in Q_s$.

For a term $t \in T_\Sigma$ and state $\alpha \in Q$, we write $\alpha :=_{\mathcal{A}} t$ iff (1) $t =_E f(u_1, \dots, u_n)$ and there is a rule $\alpha :=_{\mathcal{A}} f(\beta_1, \dots, \beta_n)$ in Δ such that $\beta_i :=_{\mathcal{A}} u_i$ for $1 \leq i \leq n$, or (2) there is a rule $\alpha :=_{\mathcal{A}} \beta$ in Δ and $\beta :=_{\mathcal{A}} t$. A term $t \in T_{\Sigma, s}$ is accepted by \mathcal{A} if the complete set of states that generate t , $\text{gen}_{\mathcal{A}}(t) = \{\alpha \in Q_s \mid \alpha :=_{\mathcal{A}} t\}$, is a model of ϕ_s , i.e. $\text{gen}_{\mathcal{A}}(t) \models \phi_s$. Boolean formulae are evaluated using their standard interpretations:

$$P \models q \text{ if } q \in P, \quad P \models \phi_1 \vee \phi_2 \text{ if } P \models \phi_1 \text{ or } P \models \phi_2, \text{ and } P \models \neg \phi \text{ if } P \not\models \phi.$$

The language accepted by \mathcal{A} is the S -indexed family $\mathcal{L}(\mathcal{A}) = \{\mathcal{L}_s(\mathcal{A})\}_{s \in S}$ of sets of terms accepted by \mathcal{A} , i.e., $\mathcal{L}_s(\mathcal{A}) = \{t \in T_{\Sigma, s} \mid \text{gen}_{\mathcal{A}}(t) \models \phi_s\}$.

Given a PTA $\mathcal{A} = (\mathcal{E}, Q, \Phi, \Delta)$ with $\mathcal{E} = (\Sigma, E)$, we let \mathcal{A}_\emptyset denote the same PTA formed over the free theory with symbols in Σ , i.e., $\mathcal{A}_\emptyset = ((\Sigma, \emptyset), Q, \Phi, \Delta)$. By using grammar rules instead of rewrite rules in the definition of PTA, we are able to show the following for arbitrary equational theories that may be non-linear. This was proven for equational tree automata by Verma in (21) — the proof in this case is identical.

Lemma 2. *Given a PTA $\mathcal{A} = (\mathcal{E}, Q, \Phi, \Delta)$ with $\mathcal{E} = (\Sigma, E)$, if $\alpha :=_{\mathcal{A}} t$, then there must be a term $u \in T_\Sigma$ such that $t =_E u$ and $\alpha :=_{\mathcal{A}_\emptyset} u$.*

The languages recognized by PTA over a theory \mathcal{E} are precisely those languages that are in the Boolean closure of regular equational tree automata languages sharing the same theory \mathcal{E} . This is an important property, because in general, equational tree automata are not closed under Boolean operations (12).

We can use PTA to reduce the CS completeness properties for a TRS $\mathcal{R} = (\Sigma, A, R)$ into the emptiness test for a PTA with the same axioms A . When A consists of any combination of associativity, commutativity, and identity axioms, the techniques from (12) can be used to check the emptiness of the corresponding PTA. It is known that the emptiness problem for PTA is decidable A contains any combination of associativity, commutativity, and identity and every associative symbol is also commutative. If A contains an associative symbol that is not commutative, the problem is not decidable. However, there is a semi-algorithm in (12) that can always show non-emptiness, and can often show emptiness in practice by using machine learning techniques.

5.2 Checking μ -Canonical Completeness

From the definition of μ -canonical completeness, we know that \mathcal{R} is not μ -canonically complete iff there is a term $t \in T_\Sigma$ that is \mathcal{R} -reducible and (\mathcal{R}, μ) -irreducible. Therefore, we can reduce the μ -canonical completeness problem to an emptiness problem of a PTA \mathcal{A} by constructing an automaton that accepts precisely those terms $t \in T_\Sigma$ that are counterexamples.

Theorem 8. *Given a left-linear TRS $\mathcal{R} = (\Sigma, A, R)$, one can effectively construct a PTA \mathcal{A}_{cc} such that \mathcal{R} is μ -canonically complete iff $\mathcal{L}(\mathcal{A}_{cc}) = \emptyset$.*

Proof. (Sketch) Let $\Sigma = (S, F, \leq)$. \mathcal{A}_{cc} is a PTA with signature $\Sigma^K = (S^K, F^K)$ as defined in Definition 5. For each sort $s \in S$, \mathcal{A}_{cc} contains a state α_s such that for each $t \in T_{\Sigma^K}$, $\alpha_s :=_{\mathcal{A}_{cc}} t$ iff $t \in T_{\Sigma, s}$. For each $k \in K$, \mathcal{A}_{cc} contains states r_k, r_k^μ for recognizing \mathcal{R} -reducible and (\mathcal{R}, μ) reducible terms respectively. The acceptance formula for each $k \in S^K$, then just becomes $\phi_k = \alpha_k \wedge r_k \wedge \neg r_k^\mu$. The full construction and correctness proof of \mathcal{A}_{cc} is in (10). \square

The algorithm for constructing the tree automaton \mathcal{A}_{cc} from a Maude specification has been implemented, and integrated into the Maude Sufficient Completeness Checker (11). By using the tool to check the μ -canonical completeness of

the FACTORIAL specification given in Section 4.1, we are able to verify that it is μ -canonically complete:

```
Maude> (ccc FACTORIAL .)
Checking canonical completeness of FACTORIAL ...
Success: FACTORIAL is canonically complete.
```

By using the tool to check the INF-LIST specification, we find a counterexample showing that the specification is not μ -canonically complete:

```
Maude> (ccc INF-LIST .)
Checking canonical completeness of INF-LIST ...
Failure: The term 0 : first(0, []) is a counterexample that is
mu-irreducible, but reducible under ordinary rewriting.
```

5.3 Checking μ -Semantic Completeness

Since we were able to check the μ -canonical completeness of a left-linear specification \mathcal{R} using the results in the previous section, using the results in Theorem 5, the μ -semantic completeness of specifications can be mechanically checked by showing: (1) μ -canonical completeness with the checker in the previous section; (2) μ -terminating with a CS termination tool such as (4; 8; 17); and (3) confluence and sort-decreasingness with a tool such as the Maude Church-Rosser checker. This allows us to show that, for example, the FACTORIAL specification is μ -semantically complete.

5.4 Checking μ -Sufficient Completeness

Using our definition of μ -sufficient completeness, we are able to extend the Maude Sufficient Completeness Checker in (11) to the CS case.

Theorem 9. *Given a left-linear TRS \mathcal{R} that is ground μ -weakly normalizing and ground μ -sort-decreasing, one can construct a PTA \mathcal{A}_{SC} such that \mathcal{R} is μ -sufficiently complete relative to C iff $\mathcal{L}(\mathcal{A}_{\text{SC}}) = \emptyset$.*

We have also implemented an algorithm for constructing the automaton \mathcal{A}_{SC} from a CS Maude specification automatically. In this case, the checker succeeds on the FACTORIAL example, as expected:

```
Maude> (mu-scc FACTORIAL .)
Checking the mu-sufficient completeness of FACTORIAL ...
Success: FACTORIAL is mu-sufficiently complete assuming that it is
ground mu-weakly normalizing and ground mu-sort-decreasing.
```

Running the checker on the INF-LIST example yields an error:

```
Maude> (mu-scc INF-LIST .)
Checking the mu-sufficient completeness of INF-LIST ...
Failure: The term sel(0, []) is an mu-irreducible term with sort
Nat? in INF-LIST with defined symbols in replacement positions.
```

It turns out that the rewrite system given in (16) was missing equations for defining `sel` and `first` when the second argument was the empty list. If we add the equations “`sel(M, []) = none`” and “`first(M, []) = []`” to the Maude specification, the μ -sufficient completeness check succeeds.

6 Related Work and Conclusions

An earlier paper by Lucas (14) has a section on relating the \mathcal{R} and (\mathcal{R}, μ) -canonical forms. In one of the results, a replacement map $\mu_{\mathcal{R}}^B$ is constructed from \mathcal{R} and the subset of symbols $B \subseteq F$, and results show that \mathcal{R} is μ -canonically complete if the (\mathcal{R}, μ) -irreducible terms are in T_B , and $\mu \supseteq \mu_{\mathcal{R}}^B$. This condition is sufficient to show that the FACTORIAL example is μ -canonically complete. However it is easy to give examples where \mathcal{R} is μ -canonically complete, but $\mu \not\supseteq \mu_{\mathcal{R}}^B$. Since we have now a decision procedure for μ -canonical completeness, by varying the replacement map μ , one can use our results to find all *minimal* replacement maps μ for which \mathcal{R} is μ -canonically complete.

It would be useful to investigate the relationships between the work we have presented here and infinite rewriting and infinite normal forms, e.g., (1; 3), which has been extended to the CS in (15). In particular, it seems interesting to investigate the relations between algebras of finite and infinite terms, and the extension of sufficient completeness to infinite normal forms.

We have proposed a new model-theoretic semantics for order-sorted CS specifications in the form of the μ -canonical term algebra $\text{Can}_{\mathcal{R}}^{\mu}$. And we have investigated three notions of CS completeness: (1) μ -canonical completeness with respect to canonical forms; (2) μ -semantic completeness with respect to equational deduction; and (3) μ -sufficient completeness with respect to constructors. We have also proposed and implemented decision procedures based on propositional tree automata that, under reasonable assumptions on the CS specification (which can be discharged by other existing tools), ensure that it satisfies the different μ -completeness properties. These results provide new ways of reasoning formally about CS equational programs, not only to allow a programmer to check that his/her program behaves as desired, but also to prove properties: for example it *is* sound to use an inductive theorem prover to reason about a μ -semantically complete CS program, whereas in general such reasoning may be unsound, since $\text{Can}_{\mathcal{R}}^{\mu}$ may not satisfy the equations of \mathcal{R} and may have “junk” data outside the image from the initial algebra.

We think that it would be useful to extend the concepts and results presented here to: (1) more general conditional CS specifications in membership equational logic (19); (2) CS specifications with non-left-linear rules, for which the tree automata techniques proposed in (13) could be quite useful; and (3) infinite μ -normal forms and infinitary rewriting, as discussed above.

Acknowledgements. The authors would like to thank Salvador Lucas and the anonymous referees for comments that helped to improve the paper.

References

- [1] Boudol, G.: Computational semantics of term rewriting systems. *Algebraic methods in semantics*, 169–236 (1986)
- [2] Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: *All About Maude*. Springer LNCS vol. 4350 (To appear 2007)
- [3] Dershowitz, N., Kaplan, S., Plaisted, D.A.: Rewrite, rewrite, rewrite, rewrite, rewrite, *Theor. Comput. Sci.* 83(1), 71–96 (1991)
- [4] Durán, F., Lucas, S., Meseguer, J., Marché, C., Urbain, X.: Proving termination of membership equational programs. In: *Proc. of PEPM*, pp. 147–158. ACM, New York (2004)
- [5] Futatsugi, K., Diaconescu, R.: *CafeOBJ Report*. World Scientific, AMAST Series (1998)
- [6] Futatsugi, K., Goguen, J., Jouannaud, J.-P., Meseguer, J.: Principles of OBJ2. In: *Proc. of POPL 1985*, pp. 52–66. ACM, New York (1985)
- [7] Giesl, J., Middeldorp, A.: Transformation techniques for context-sensitive rewrite systems. *J. Funct. Program* 14(4), 379–427 (2004)
- [8] 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)
- [9] Goguen, J., Diaconescu, R.: An oxford survey of order sorted algebra. *Mathematical Structures in Computer Science* 4(3), 363–392 (1994)
- [10] Hendrix, J., Meseguer, J.: On the completeness of context-sensitive order-sorted specifications. Technical Report UIUCDCS-R-2007-2812, University of Illinois (2007) Available at <http://maude.cs.uiuc.edu/tools/sccl/>
- [11] Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: *IJCAR 2006*. LNCS, vol. 4130, pp. 151–155. Springer, Heidelberg (2006)
- [12] Hendrix, J., Ohsaki, H., Viswanathan, M.: Propositional tree automata. In: *RTA 2006*. LNCS, vol. 4098, pp. 165–174. Springer, Heidelberg (2006)
- [13] Jacquemard, F., Rusinowitch, M., Vigneron, L.: Tree automata with equality constraints modulo equational theories. In: *IJCAR 2006*. LNCS, vol. 4130, pp. 557–571. Springer, Heidelberg (2006)
- [14] Lucas, S.: Context-sensitive computations in fonctionnal and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1) (1998)
- [15] Lucas, S.: Transfinite rewriting semantics for term rewriting systems. In: Middeldorp, A. (ed.) *RTA 2001*. LNCS, vol. 2051, pp. 216–230. Springer, Heidelberg (2001)
- [16] Lucas, S.: Context-sensitive rewriting strategies. *Information and Computation* 178(1), 294–343 (2002)
- [17] 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)
- [18] Lucas, S.: Proving termination of context-sensitive rewriting by transformation. *Information and Computation* 204(12), 1782–1846 (2006)
- [19] 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)

- [20] Ohsaki, H.: Beyond regularity: Equational tree automata for associative and commutative theories. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 539–553. Springer, Heidelberg (2001)
- [21] Verma, K.N.: Two-way equational tree automata for AC-like theories: Decidability and closure properties. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 180–196. Springer, Heidelberg (2003)
- [22] Zantema, H.: Termination of context-sensitive rewriting. In: Comon, H. (ed.) Proc. of RTA'97. LNCS, vol. 1232, pp. 172–186. Springer, Heidelberg (1997)

KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis*

Mark Hills and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801
{mhills,grosu}@cs.uiuc.edu
<http://fsl.cs.uiuc.edu>

Abstract. This paper presents KOOL, a concurrent, dynamic, object-oriented language defined in rewriting logic. KOOL has been designed as an experimental language, with a focus on making the language easy to extend. This is done by taking advantage of the flexibility provided by rewriting logic, which allows for the rapid prototyping of new language features. An example of this process is illustrated by sketching the addition of synchronized methods. KOOL also provides support for program analysis through language extensions and the underlying capabilities of rewriting logic. This support is illustrated with several examples.

Keywords: object-oriented languages, programming language semantics, term rewriting, rewriting logic, formal analysis.

1 Introduction

Language design is both an art and a science. Along with formal tools and notations to address the science, it is important to have good support for the “art”: tools that allow the rapid prototyping of language features, allowing new features to be quickly developed, tested, and refined (or discarded). Rewriting logic, briefly introduced in Section 2, provides a good environment for addressing both the art and the science: formal tools for specifying language semantics and analyzing programs, plus a flexible environment for adding new features that automatically provides language interpreters. As an example of the capabilities of this model of language design, we have created KOOL, a concurrent, dynamic, object-oriented language built using rewriting logic. The KOOL language and environment are described in 3, since KOOL is focused on language experimentation, Section 4 illustrates this by sketching the addition of synchronized methods, a concurrency-related feature borrowed from the Java language 7. KOOL also inherits support for analysis from rewriting logic, which has been enhanced with language constructs and runtime support. This is explored in Section 5. Section 6 concludes, summarizing and discussing some related work.

* Supported by NSF CCF-0448501 and NSF CNS-0509321.

A sister paper [10] presents more detailed information about formal analysis of KOOL programs, including a detailed look at how design decisions in rewriting logic definitions of object oriented languages impact analysis performance.

2 Rewriting Logic

Rewriting logic [12,13,14,15] is a computational logic built upon equational logic which provides support for concurrency. In equational logic, a number of *sorts* (types) and *equations* are defined. The equations specify which terms are considered to be equal. All equal terms can then be seen as members of the same equivalence class of terms, a concept similar to that from the λ calculus with equivalence classes based on α and β equivalence. Rewriting logic provides *rules* in addition to equations, used to transition between equivalence classes of terms. This allows for concurrency, where different orders of evaluation could lead to non-equivalent results, such as in the case of data races. The distinction between rules and equations is crucial for analysis, since terms which are equal according to equational deduction can all be collapsed into the same analysis state. Rewriting logic is connected to term rewriting in that all the equations and rules of rewriting logic, of the form $l = r$ and $l \Rightarrow r$, respectively, can be transformed into term rewriting rules by orienting them properly (necessary because equations can be used for deduction in either direction), transforming both into $l \rightarrow r$. This provides a means of taking a definition in rewriting logic and using it to “execute” a term using standard term rewriting techniques.

3 KOOL

KOOL is a concurrent, dynamic, object-oriented language, loosely inspired by, but not identical to, the Smalltalk language [6]. KOOL includes support for standard imperative features, such as assignment, conditionals, and loops with break and continue. KOOL also includes support for many familiar object-oriented features: all values are objects; all operations are carried out via message sends; message sends use dynamic dispatch; single inheritance is used, with a designated root class named `Object`; methods are all public, while fields are all private outside of the owning object; and scoping is static, yet declaration order for classes and methods is unimportant. KOOL allows for the run-time inspection of object types via a `typecase` construct, and includes support for exceptions with a standard `try/catch` mechanism.

3.1 KOOL Syntax

The syntax of KOOL is shown in Figure 1. The lexical definitions of literals are not included in the figure to limit clutter, but are standard (for instance, booleans include both `true` and `false`, strings are surrounded with double quotes, etc). Most message sends are specified in a Java-like syntax; those representing binary operations can also be used infix (`a + b` desugars to `a.(+)(b)`), with these

<i>Program</i>	$P ::= C^* E$
<i>Class</i>	$C ::= \text{class } X \text{ is } D^* M^* \text{ end} \mid \text{class } X \text{ extends } X' \text{ is } D^* M^* \text{ end}$
<i>Decl</i>	$D ::= \text{var } \{X, \}^+ ;$
<i>Method</i>	$M ::= \text{method } X \text{ is } D^* S \text{ end} \mid \text{method } X (\{X', \}^+) \text{ is } D^* S \text{ end}$
<i>Expression</i>	$E ::= X \mid I \mid F \mid B \mid Ch \mid Str \mid (E) \mid \text{new } X \mid \text{new } X (\{E, \}^+) \mid$ $\text{self} \mid E X_{op} E' \mid E.X(\)^? \mid E.X(\{E, \}^+) \mid \text{super}(\) \mid$ $\text{super}.X(\)^? \mid \text{super}.X(\{E, \}^+) \mid \text{super}(\{E, \}^+) \mid \text{primInvoke}(\{E, \}^+)$
<i>Statement</i>	$S ::= E \leftarrow E' ; \mid \text{begin } D^* S \text{ end} \mid \text{if } E \text{ then } S \text{ (else } S')^? \text{ fi} \mid$ $\text{try } S \text{ catch } X S \text{ end} \mid \text{throw } E ; \mid \text{while } E \text{ do } S \text{ od} \mid$ $\text{for } X \leftarrow E \text{ to } E' \text{ do } S \text{ od} \mid \text{break} ; \mid \text{continue} ; \mid$ $\text{return } (E)^? ; \mid S S' \mid E ; \mid \text{assert } E ; \mid X ; \mid \text{spawn } E ; \mid$ $\text{acquire } E ; \mid \text{release } E ; \mid \text{typecase } E \text{ of } C_s^+ \text{ (else } S)^? \text{ end}$
<i>Case</i>	$C_s ::= \text{case } X \text{ of } S$

$X \in \text{Name}, I \in \text{Integer}, F \in \text{Float}, B \in \text{Boolean}, Ch \in \text{Char}, Str \in \text{String}, X_{op} \in \text{Operator Names}$

Fig. 1. KOOL Syntax

infix usages all having the same precedence and associativity. Finally, semicolons are used as statement terminators, not separators, and are only needed where the end of a statement may be ambiguous – at the end of an assignment, for instance, or at the end of each statement inside a branch of a conditional, but not at the end of the conditional itself, which ends with `fi`.

To get a feel for the language, two sample class definitions are presented in Figure 2. These definitions provide a simple example of inheritance and calls to super-methods using a familiar Point/ColorPoint example. Class `Point` represents a point in 2D space, with `x` and `y` coordinates, and implicitly inherits from (extends) class `Object`.

Class `ColorPoint` explicitly extends class `Point`, adding a new variable `c` to represent the color of the point. The constructor for `ColorPoint` calls its

```

class Point is
  var x,y;

  method Point(inx, iny) is
    x <- inx; y <- iny;
  end

  method toString is
    return ("x = " + x.toString() + " and y = "
           + y.toString());
  end
end

class ColorPoint extends Point is
  var c;

  method ColorPoint(inx, iny, inc) is
    super(inx,iny); c <- inc;
  end

  method toString is
    return (super.toString() + " and c = "
           + c.toString());
  end
end

```

Fig. 2. Inheritance and Built-ins in KOOL

parent constructor, passing the `x` and `y` coordinates, while the version of `toString` defined in `ColorPoint` also uses `super`, here to invoke the parent version of the `toString` method. `+` is defined in the `String` class as string concatenation.

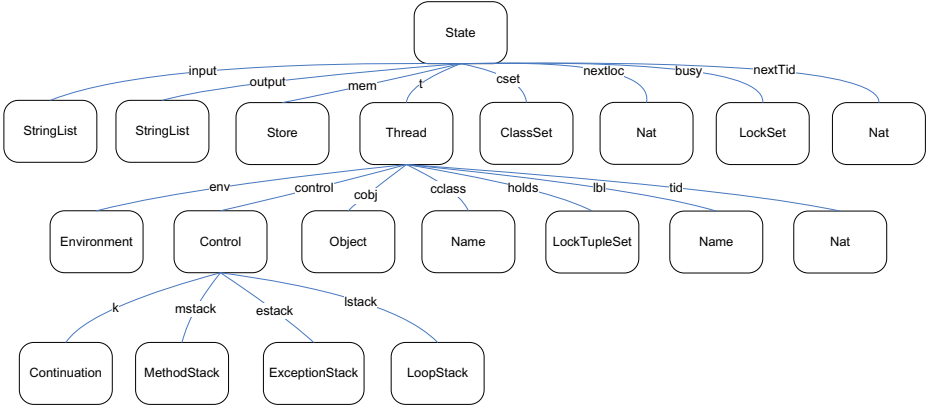


Fig. 3. KOOL State Infrastructure

3.2 KOOL Semantics

The semantics of KOOL is defined using Maude [2,3], a high-performance language and engine for rewriting logic. The current program is represented as a “soup” (multiset) of nested terms representing the current computation, memory, locks held, etc. A visual representation of this term, the state infrastructure, is shown in Figure 3. Here, each box represents a piece of information stored in the program state, with the text in each box indicating the information’s *sort*, such as `Name` or `LockSet`. Edges between boxes represent the names used to reference the information, such as `cclass` (the current class context) or `busy` (a set of all locks held by any thread in the program), and are defined as operations from the boxed to the containing sort (i.e., `cclass` is an operation from `Name` to `Thread`, so it can be treated as thread information). Information stored in the state can be nested: `State` contains at least one `Thread`, accessed with `t`, which contains `Control` information, accessed with `control`. The most important piece of information is the `Continuation`, located in `Control` and named `k`, which is a first-order representation of the current computation, made up of a list of instructions separated by `->`. The continuation can be seen as a stack, with the current instruction at the left and the remainder (continuation) of the computation to the right. This *continuation-based* methodology is described in more detail in papers about the rewriting logic semantics project [14,15].

Figure 4 shows examples of Maude equations and rules included in the KOOL semantics. The first three equations (shown with `eq`) process a conditional. The first indicates the value of the guard expression `E` must be computed before a

branch statement (S or S') is evaluated; to do this, E is put before the branches on the continuation, with the branches saved for later use by putting them into an `if` continuation item. The second and third execute the appropriate branch based on whether the guard evaluated to `true` or `false`. The fourth, a conditional rule (represented with `cr1`), represents the lookup of a memory location. The rule states that, if the next computation step in this thread is to look up the value at location L , and if that value is V (`:=` binds V to the result of reducing $\text{Mem}[L]$, the memory lookup operation), and if V is not undefined (i.e. L and V are actually in Mem), the result of the computation is the value V . This must be a rule, since memory reads and writes among different threads could lead to non-equivalent behaviors. CS and TS match the unreferenced parts of the control and thread state, respectively, while K represents the rest of the computation in this thread. Note that, since the fourth rule represents a side-effect, it can only be applied when it is the next computation step in the thread (it is at the head of the continuation), while the first three, which don't involve side-effects, can be applied at any time.

```

eq stmt(if E then S else S' fi) = exp(E) -> if(S,S') .
eq val(primBool(true)) -> if(S,S') = stmt(S) .
eq val(primBool(false)) -> if(S,S') = stmt(S') .

cr1 t(control(k(lookup(L) -> K) CS) TS) mem(Mem) =>
  t(control(k(val(V) -> K) CS) TS) mem(Mem)
if V := Mem[L] /\ V /= undefined .
  
```

Fig. 4. Sample KOOL Rules

3.3 KOOL Implementation

There is an implementation of KOOL available at our website [11], as well as a web-based interface to run and analyze KOOL programs such as those presented here. The website also contains current information about the language, which is constantly evolving. A companion technical report [9] explains the syntax and semantics of KOOL in more detail.

KOOL programs are generally run using the `runkool` script, since running a KOOL program involves several steps and tools, and since programs can be run in different modes (execution, search, and model checking, each with various

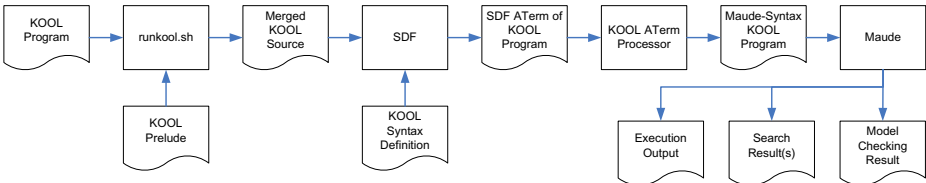


Fig. 5. KOOL Program Evaluation/Analysis

options). First, the KOOL prelude, a shared set of classes for use in user programs, is added to the input program. This program is then parsed using the SDF parser [19], which takes the program text and a syntax definition file as input. The parser produces a file in ATerm format, an SDF format used to represent the abstract syntax tree. A pretty printer then converts this into Maude, using prefix versions of the operators to prevent parsing difficulties. Finally, Maude is invoked by `runkool` with the language semantics and the Maude-format program, generating the result based on the execution mode. A graphical view of this process is presented in Figure 5.

4 Extending KOOL

Because of its use in our language research and in teaching, the KOOL system has been designed to be extensible. To illustrate the extension process at a high level, an example extension, `synchronized` methods, is presented here. Further details, including a full implementation, are available at the KOOL website [11].

The KOOL language has a fairly simple model of concurrency based on threads, which each contain their own continuation and execution context (shown in Figure 3). Each program starts with one thread. The `spawn` statement can then be used to create more threads, which execute independently of one another. All mutual exclusion is handled with object-level locks, acquired using `acquire` and released with `release`. Objects can only be locked by one thread at a time, although that thread may lock the same object more than once. If this happens, the thread needs to release an equivalent number of locks on the object before it can be `acquire`'d by another thread. `synchronized` methods, similar to those in Java, would provide a higher-level abstraction over these locking primitives. In Java, methods tagged with the `synchronized` keyword implicitly lock the object that is the target of the method call, allowing only one thread to be active in all `synchronized` methods on a given object at a time. We will assume the same semantics for KOOL.

The syntax changes to add `synchronized` methods are minor: the keyword needs to be added to the method syntax, which then also needs to be reflected in the Maude-level syntax for KOOL. In SDF, two context-free definitions for Method nonterminals need to be added, one for `synchronized` methods with parameters, the other for those without. Similar changes need to occur in Maude, where one additional syntax operation needs to be added. Finally, the pretty printer that translates from SDF ATerms into Maude needs to be modified to handle the two new SDF definitions; this change is fairly mechanical, and methods to eliminate the need for this are being investigated.

The changes to the semantics are obviously more involved. In KOOL, as in Java, `synchronized` methods should work as follows:

- a call to a `synchronized` method should implicitly acquire a lock on the message target before the method body is executed;
- a return from a `synchronized` method should release this lock;

- additional calls to synchronized methods on the same target should be allowed in the same thread;
- exceptional returns from methods should release any locks implicitly acquired when the method was called.

A quick survey of these requirements shows that adding **synchronized** methods will change more than the message send semantics – the semantics for exceptions will need to change as well, to account for the last requirement.

To handle the first requirement, a new lock can be acquired on the **self** object at the start of any **synchronized** method simply by adding a lock acquisition to the start of the method body, which can be done when the method definition is processed. Lock release cannot be handled similarly, though, since there may be multiple exits from a method, including **return** statements and exceptional returns. This means that locks acquired on method entry will need to be tracked so they can be properly released on exit. This can be accomplished by recording the lock information in the method and exception stacks (**mstack** and **estack** in Figure 3) when the lock is acquired, since these stacks are accessed in the method return and exception handling semantics. With this in place, the second and fourth requirements can be handled by using this recorded information to release the locks on method return or when an exception is thrown. Finally, the third point is naturally satisfied by the existing concurrency semantics, which allow multiple locks on the same object (here, **self**) by the same thread. Overall, adding synchronized methods to the KOOL semantics requires:

- 2 modified operators (to add locks to the two stack definitions),
- 4 modified equations (two for method return, two for exception handling),
- 4 new operators (to record locks in the stacks, to release all recorded locks),
- 6 new equations (to record locks in the stacks, to release all recorded locks).

With these changes, KOOL includes 243 operators, 334 equations, and 15 rules.

5 Analyzing KOOL Programs

KOOL program analysis is based around the underlying functionality provided by Maude and rewriting logic, including the ability to perform model checking based on LTL formulae and program states and the ability to perform a breadth-first search of the program state space. These capabilities have been extended with additions to the KOOL language, including assertion checking capabilities that interact with the model checker and program labels that can be used in LTL formulae to check progress (or the lack thereof) between program points.

5.1 Breadth-First Search

The thread game is a concurrency problem defined as follows: take a single variable, say x , initialized to 1. In two threads, repeat the assignment $x \leftarrow x + x$

forever. In another thread, output the value of x . What values is it possible to output? As has been proved [16], it is possible to output any natural number ≥ 1 . A KOOL version of the thread game is shown in Figure 6.

To check if a specific value can be output, one could run the program repeatedly, or try model checking. However, with the program's infinite state space and nondeterministic behavior, this may never yield the desired result. Maude's search capability can be used, though, either to enumerate possible values (obviously not all possible values here) or to search for a specific value. For instance, searching for 10 yields a result, indicating that 10 is one of the possible output values.

Search can also be useful when adding new language features. For instance, an example of `synchronized` methods in KOOL is shown in Figure 7. Here, class `WriteNum` contains two `synchronized` methods. When the `write` method is called, the starting value of the number stored in member variable `num` is written to the console, some simple arithmetic operations are performed on it, and then the final value is written. The `set` method assigns a new value to `num`. Since both methods are marked `synchronized`, it should be the case that, for any given object, once one thread is executing either method, another thread that tries to execute either will wait. To test this, the `Driver` class creates a new object of class `WriteNum`, spawns one call to `write`, creating a new thread, modifies the value stored in the object using `set`, and then creates a second thread, also calling `write`. Using search to determine possible program outputs reveals that there are only two possible solutions, with the call to `set` either occurring before the first spawned thread runs (with output "Start:", "20", "End:", "22", "Start:", "22", "End:", "24"), or after it completes (printing "Start:", "10", "End:", "12", "Start:", "20", "End:", "22").

```
class ThreadGame is
  var x;

  method ThreadGame is
    x <- 1;
  end

  method Add is
    while true do x <- x + x; od
  end

  method Run is
    spawn(self.Add); spawn(self.Add);
    console << x;
  end
end
(new ThreadGame).Run
```

Fig. 6. Thread Game

```
class WriteNum is
  var num;

  method WriteNum(n) is
    num <- n;
  end

  synchronized method set(n) is
    num <- n;
  end

  synchronized method write is
    console << "Start:" << num;
    self.set(num + 10);
    self.set(num - 8);
    console << "End:" << num;
  end
end

class Driver is
  method run is
    var w1;
    w1 <- new WriteNum(10);
    spawn(w1.write);
    w1.set(20);
    spawn(w1.write);
  end
end

(new Driver).run
```

Fig. 7. Synchronized Methods

By contrast, with the `synchronized` keywords removed, there are 470 solutions, corresponding to all possible orderings of output based on various interleavings of the main thread with the two spawned threads.

5.2 Model Checking

Programs in KOOL can take advantage of Maude’s model-checking capabilities. The Maude model checker uses LTL formulae, which are written against the state infrastructure. Since the state can be very complex, KOOL allows *labels*, which can be referenced in LTL formulae, to be included in the program source. For example, Figure 8 contains a KOOL fragment of the Dining Philosophers problem. Each `Philosopher` will try to lock left and right `Forks` before eating, releasing them when finished.

We can check for deadlock freedom by verifying that reaching label `hungry`: implies always eventually reaching label `eating`: (always eventually acquiring both locks). Results for model checking a deadlocking and a fixed version of the problem with 5, 6, and 8 philosophers are shown in Figure 9. More details about model checking and search in KOOL, including further discussion of labels, additional examples of search, additional performance measurements, and an investigation of the impact of language design on analysis performance can be found in a related paper [10].

```
class Fork is end
class Philosopher is
  method Run(id,left,right) is
    while (true) do
      hungry:
        acquire left; acquire right;
      eating:
        release left; release right;
    od
  end
end
```

Fig. 8. Philosophers and Forks

Philosophers	# of States	Find Counterexample/s	Prove Deadlock Freedom/s
5	634	1.989	23.014
6	2943	8.444	130.174
8	63505	278.276	4975.583

3.40 GHz P4, 2 GB RAM, OpenSuSE 10.2, kernel 2.6.16.27-0.6-smp, Maude 2.2.

Fig. 9. Dining Philosophers Verification Time

6 Conclusions and Related Work

In this paper we have presented the KOOL language as a concrete application of rewriting logic to language design and program analysis, illustrating the process of language extension and highlighting the provided analysis capabilities. KOOL has been used as a basis for both teaching [17] and research in language semantics, design, analysis, and verification. The evolving nature of KOOL seems to make it especially appealing for classroom use, where student projects have included adding reflection and additional concurrency features.

There is a large volume of work related to defining programming languages using executable techniques. Probably the most related is the JavaFAN project, which defined formal analysis tools for both the Java language [4] and JVM bytecode [5]. In contrast to our work, which has focused on the design of languages and feature prototyping, JavaFAN has focused on formal methods applications, with the goal of being a competitive Java formal analysis tool. The published work on Java [4], for instance, is a 4 page tools paper that provides a short summary of the rewriting logic definition of Java, focusing instead on a description of the tool, leaving the language definition unpublished. The languages are also quite different, leading to different challenges in implementation (such as KOOL's use of Smalltalk-like primitives, here fragments of Maude, to implement low-level operations).

Functional languages defined in Maude include CML [1] and Eden [8]. Work on the latter, a parallel variant of Haskell, has also looked to Maude as a language experimentation environment, using the module system to vary the degree of parallelism and the scheduling algorithm. Still in the realm of term rewriting, the ASF+SDF project [19] has focused on language prototyping and definition. More related work can be found in papers on the rewriting logic semantics of programming languages [14,15] and the K language definition technique [18].

Acknowledgments. We thank the anonymous reviewers for their helpful comments, which have improved the quality of this paper.

References

1. Chalub, F., Braga, C.: A Modular Rewriting Semantics for CML. In: Proceedings of the 8th. Brazilian Symposium on Programming Languages (May 2004)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: specification and programming in rewriting logic. *Theoretical Computer Science* 285, 187–243 (2002)
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The Maude 2.0 System. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
4. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal Analysis of Java Programs in JavaFAN. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 501–505. Springer, Heidelberg (2004)
5. Farzan, A., Meseguer, J., Roşu, G.: Formal JVM Code Analysis in JavaFAN. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 132–147. Springer, Heidelberg (2004)
6. Goldberg, A., Robson, D.: Smalltalk-80: the language and its implementation. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (1983)
7. Gosling, J., Joy, B., Steele, G.: The Java Language Definition. Addison-Wesley, Reading (1996)
8. Hidalgo-Herrero, M., Verdejo, A., Ortega-Mallén, Y.: Using Maude and its strategies for defining a framework for analyzing Eden semantics. In: Proceedings of WRS'06, Elsevier, Amsterdam (To appear 2006)

9. Hills, M., Roşu, G.: KOOL: A K-based Object-Oriented Language. Technical Report UIUCDCS-R-2006-2779, University of Illinois at Urbana-Champaign (2006)
10. Hills, M., Roşu, G.: On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In: Proceedings of FMOODS'07. LNCS, Springer, Heidelberg (To appear 2007)
11. Hills, M., Rosu, G.: KOOL Language Homepage <http://fsl.cs.uiuc.edu/KOOL>
12. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. *Theoretical Computer Science* 285, 121–154 (2002)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
14. Meseguer, J., Roşu, G.: Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 1–44. Springer, Heidelberg (2004)
15. Meseguer, J., Roşu, G.: The rewriting logic semantics project. *Theoretical Computer Science* (To appear 2007)
16. Moore, J.S.: <http://www.cs.utexas.edu/users/moore/publications/thread-game.html>
17. Roşu, G.: Lecture notes of course on Programming Language Design. Dept. of Computer Science, UIUC (2006) <http://fsl.cs.uiuc.edu/index.php/CS422>
18. Roşu, G.: K: a Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Department of Computer Science, University of Illinois at Urbana-Champaign (2006)
19. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS* 24(4), 334–368 (2002)

Simple Proofs of Characterizing Strong Normalization for Explicit Substitution Calculi

Kentaro Kikuchi

RIEC, Tohoku University
Katahira 2-1-1, Aoba-ku, Sendai 980-8577, Japan
`kentaro@nue.riec.tohoku.ac.jp`

Abstract. We present a method of lifting to explicit substitution calculi some characterizations of the strongly normalizing terms of λ -calculus by means of intersection type systems. The method is first illustrated by applying to a composition-free calculus of explicit substitutions, yielding a simpler proof than the previous one by Lengrand et al. Then we present a new intersection type system in the style of sequent calculus, and show that it characterizes the strongly normalizing terms of Dyckhoff and Urban’s extension of Herbelin’s explicit substitution calculus.

1 Introduction

Explicit substitution calculi were introduced for improving implementations of functional programming languages based on λ -calculus. In those calculi, substitution is not treated as a meta-operation on terms but rather as a new operator in the language. Since operational properties of substitution are studied in the object-level, unexpected behavior at the time of implementation is minimized. Also, a fine-grained control of substitution is made available; for instance, we may delay substitutions in order to avoid unnecessary duplication of information.

When augmenting λ -calculus with explicit substitutions, the evaluation process is refined by reduction rules to deal with substitutions. This suggests that reduction properties of explicit substitution calculi may vary from those of the original λ -calculus. In fact, as shown by Melliès [17], there are simply typed λ -terms that are not strongly normalizing when evaluated by the reduction rules of the explicit substitution calculus in [1].

In this paper we first study a composition-free calculus of explicit substitutions $\lambda\mathbf{x}$ [5], in which strong normalization holds for simply typed terms. In [9], Dougherty and Lescanne presented intersection type assignment systems for $\lambda\mathbf{x}$, and showed that the terms typable in one of their systems are strongly normalizing. For the original λ -calculus, the converse also holds, i.e., all strongly normalizing terms are typable in an intersection type system [19]. The system in [9], however, does not satisfy this property. Then an extended system was developed in [16] where the typable terms coincide with the strongly normalizing ones. In the first half of this paper, we give a much simpler proof of this result than the one in [16], illustrating how to lift the characterization result for the original λ -calculus to an explicit substitutions setting.

In the latter half of the paper, we apply our method to an explicit substitution calculus studied in [11,15]. This calculus is to sequent calculus what λx -calculus is to natural deduction. Simply typed terms of the calculus correspond to proofs in Herbelin's sequent calculus [12], and the reduction rules correspond to cut-elimination steps in the sequent calculus. Although a classical variant of Herbelin's calculus was also developed in [7], we consider in this paper the calculus of [11] because it is closer to the original λ -calculus and so better for a first study of intersection type assignment systems based on sequent calculus. We introduce a new type assignment system for the explicit substitution calculus, and show that the typable terms coincide with the strongly normalizing ones. Our proof method successfully applies to the calculus, while the method in [16] seems difficult to apply.

Strong normalization proofs for typable terms in [9,16] use a variant of the reducibility method, but they also rely on two lemmas in Section 3 of [9] whose proofs are rather complicated. On the other hand, our proof of strong normalization relies on a theorem that was proved in [4] using recursive path ordering [8] and an encoding of λ -terms with explicit substitutions into a first-order rewriting system. Thus, in our proof, the complicated part is solved by a well-established result in term rewriting. This method was used for the simply typed case of the explicit substitution calculus in [11] (see also the remark after Theorem 3).

To prove that all strongly normalizing terms are typable, we develop a novel technique. The method in [16] uses a specific perpetual strategy or an inductive characterization of strongly normalizing terms in the style of [20,6]. However, such a perpetual strategy or an inductive characterization is difficult to spell out when considering a more complex explicit substitution calculus than λx . The idea behind our technique is instead that strongly normalizing terms are closed under x -conversion as far as decent terms are concerned (decent terms are terms in which every substitution body is strongly normalizing). This was pointed out in [14] for λx^- (λx with restricted garbage collection), and in [15] for the explicit substitution calculus of [11]. For an inductive argument to work, we introduce the notion of typably decent terms, which are defined as the terms in which every substitution body is typable. It is then sufficient to show that typable terms are closed under x -conversion as far as typably decent terms are concerned.

The paper is organized as follows. In Section 2 we introduce λx -calculus. In Section 3 we characterize the strongly normalizing λx -terms by an intersection type system in [16]. In Section 4 we introduce $\bar{\lambda} x$ -calculus. In Section 5 we characterize the strongly normalizing $\bar{\lambda} x$ -terms by a new intersection type system.

To save space we omit some proofs in Section 5, but a full version with all proofs is available at <http://www.nue.riec.tohoku.ac.jp/user/kentaro/>.

2 λx -Calculus

In this section we recall the definition and some properties of λx -calculus [5,3]. This calculus is known as the simplest explicit substitution calculus; it is up to α -conversion and uses the minimal apparatus for substitution.

Table 1. $\lambda\mathbf{x}$ -calculus

$M, N ::= x \mid MN \mid \lambda x.M \mid M\langle x := N \rangle$	
$(Beta)$	$(\lambda x.M)N \rightarrow M\langle x := N \rangle$
(App)	$(MM')\langle x := N \rangle \rightarrow M\langle x := N \rangle M'\langle x := N \rangle$
(Abs)	$(\lambda y.M)\langle x := N \rangle \rightarrow \lambda y.M\langle x := N \rangle$
(Var)	$x\langle x := N \rangle \rightarrow N$
(gc)	$M\langle x := N \rangle \rightarrow M \quad \text{if } x \notin FV(M)$

The syntax and the reduction rules of $\lambda\mathbf{x}$ -calculus are given in Table 1. The set of terms is denoted by $\mathcal{T}_{\lambda\mathbf{x}}$ and they are called $\lambda\mathbf{x}$ -terms. In $M\langle x := N \rangle$, $\langle x := N \rangle$ is called an *explicit substitution* or simply substitution and N is called the *body* of the substitution. The notions of free and bound variables are defined as usual, with an additional clause that the variable x in $M\langle x := N \rangle$ binds the free occurrences of x in M . The set of free variables of a $\lambda\mathbf{x}$ -term M is denoted by $FV(M)$. The symbol \equiv denotes syntactical equality modulo α -conversion.

The notion of $\lambda\mathbf{x}$ -reduction is defined by the contextual closures of all reduction rules in Table 1. We use $\rightarrow_{\lambda\mathbf{x}}$ for one-step reduction, $\stackrel{+}{\rightarrow}_{\lambda\mathbf{x}}$ for its transitive closure, and $\stackrel{*}{\rightarrow}_{\lambda\mathbf{x}}$ for its reflexive transitive closure. The set of $\lambda\mathbf{x}$ -terms that are strongly normalizing with respect to $\lambda\mathbf{x}$ -reduction is denoted by $\mathcal{SN}_{\lambda\mathbf{x}}$. These kinds of notations are also used for the notions of other reductions introduced in this paper.

The subcalculus of $\lambda\mathbf{x}$ without the rule $(Beta)$ is denoted by \mathbf{x} . This subcalculus has the following properties [3].

Proposition 1. *The subcalculus \mathbf{x} is strongly normalizing and confluent.*

Proof. Strong normalization is shown by defining a map $h : \mathcal{T}_{\lambda\mathbf{x}} \rightarrow \mathbb{N}$ which decreases on \mathbf{x} -reduction; define

$$\begin{aligned} h(x) &=_{def} 1 & h(MN) &=_{def} h(M) + h(N) + 1 \\ h(\lambda x.N) &=_{def} h(N) + 1 & h(M\langle x := N \rangle) &=_{def} h(M) \times (h(N) + 1) \end{aligned}$$

and observe that if $M \rightarrow_{\mathbf{x}} N$ then $h(M) > h(N)$. To prove confluence, it is now sufficient to show local confluence, which is easy. \square

As a result, we can define the unique \mathbf{x} -normal form of each $\lambda\mathbf{x}$ -term.

Definition 1. *The unique \mathbf{x} -normal form of a $\lambda\mathbf{x}$ -term M is denoted by $\mathbf{x}(M)$.*

The usual λ -terms are the $\lambda\mathbf{x}$ -terms that do not contain explicit substitutions. In this paper they are called *pure λ -terms*. The β -rule on pure λ -terms is stated as $(\lambda x.M)N \rightarrow_{\beta} M[N/x]$ where $M[N/x]$ represents meta-substitution. The relation between pure λ -terms and \mathbf{x} -normal forms is as follows.

Proposition 2. *M is a pure λ -term if and only if M is in \mathbf{x} -normal form.*

Proof. The left-to-right implication is straightforward. We prove the converse by induction on the structure of M . Suppose that M is in \mathbf{x} -normal form. Then by the induction hypothesis, all subterms of M are pure λ -terms. Now, if M is not a pure λ -term, then M is of the form $P\langle x := Q \rangle$ where P, Q are pure λ -terms. In this case, M is an \mathbf{x} -redex, which is a contradiction. \square

The next proposition shows that the subcalculus \mathbf{x} correctly simulates meta-substitution on pure λ -terms.

Proposition 3. *Let M, N be pure λ -terms. Then $M\langle x := N \rangle \xrightarrow{*}_{\mathbf{x}} M[N/x]$.*

Proof. By induction on the structure of M . \square

The next lemma shows that $\lambda\mathbf{x}$ -reduction simulates β -reduction.

Lemma 1. *Let M, N be pure λ -terms. If $M \rightarrow_{\beta} N$ then $M \xrightarrow{\pm}_{\lambda\mathbf{x}} N$.*

Proof. By induction on the reduction relation \rightarrow_{β} . We consider the case where $M \equiv (\lambda x.P)Q \rightarrow_{\beta} P[Q/x] \equiv N$. Then use \rightarrow_{Beta} to create $P\langle x := Q \rangle$, and use Proposition 3 to reach $P[Q/x]$. \square

Bloo and Geuvers [4] proved the following theorem to show that $\lambda\mathbf{x}$ -calculus satisfies the PSN property. Their method appeals to recursive path ordering [8] and a first-order encoding of $\lambda\mathbf{x}$ -terms. We use the theorem to prove one direction in characterizing strongly normalizing $\lambda\mathbf{x}$ -terms by means of intersection types.

Definition 2 (Bounded terms). *The set of bounded terms, denoted $\lambda\mathbf{x}^{<\infty}$, is defined by $\lambda\mathbf{x}^{<\infty} =_{def} \{M \mid \text{for every subterm } N \text{ of } M, \mathbf{x}(N) \in \mathcal{SN}_{\beta}\}$.*

Theorem 1 ([4]). *If $M \in \lambda\mathbf{x}^{<\infty}$ then $M \in \mathcal{SN}_{\lambda\mathbf{x}}$.*

3 Characterization of Strongly Normalizing $\lambda\mathbf{x}$ -Terms

In this section we show that the strongly normalizing $\lambda\mathbf{x}$ -terms are characterized by typability in an intersection type assignment system given in [16]. To prove that the typable terms are strongly normalizing, we use Theorem 1, subject reduction (Theorem 2), and the result in [19] for ordinary λ -calculus and intersection types. (Some of the proofs of lemmas are taken from [16].) To prove the other direction, we make an inductive argument together with the preservation of types under a certain \mathbf{x} -expansion.

First, the set of types is defined by the grammar: $\sigma ::= \varphi \mid \sigma \rightarrow \sigma \mid \sigma \cap \sigma$ where φ ranges over a denumerable set of type atoms. We use letters $\sigma, \tau, \rho, \dots$ for arbitrary types. The type assignment system $\lambda\mathbf{x}_{\cap}$ is defined by the rules in Table 2. A *typing context* is defined as a finite set of pairs $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ where the variables are pairwise distinct. The typing context $\Gamma, x : \sigma$ denotes the union $\Gamma \cup \{x : \sigma\}$ where $x \notin \Gamma$ ($x \notin \Gamma$ means that x does not appear in Γ).

Table 2. The type assignment system $\lambda\mathbf{x}_\cap$

$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} (Ax)$	$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} (\rightarrow E)$	
$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau} (\rightarrow I)$	$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} (\cap I)$	$\frac{\Gamma \vdash M : \sigma_1 \cap \sigma_2}{\Gamma \vdash M : \sigma_i} (\cap E)$ where $i \in \{1, 2\}$
$\frac{\Gamma \vdash N : \sigma \quad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash M \langle x := N \rangle : \tau} (Cut)$	$\frac{\Delta \vdash N : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M \langle x := N \rangle : \tau} (K-cut)$ where $x \notin \Gamma$	

We write $\Gamma \vdash M : \sigma$ if there exists a derivation in $\lambda\mathbf{x}_\cap$ that has this judgement as its conclusion.

The original intersection type assignment system for $\lambda\mathbf{x}$ in [9] does not have the rule $(K-cut)$ and is not enough to type all strongly normalizing terms. For example, the $\lambda\mathbf{x}$ -term $z \langle y := xx \rangle \langle x := \lambda a.aa \rangle$ is strongly normalizing but not typable in the system of [9]. For more discussions, see [16, p. 29].

The system $\lambda\mathbf{x}_\cap$ has some unusual features caused by the rule $(K-cut)$. For instance, x does not necessarily appear in Γ of $\Gamma \vdash M : \tau$ even if $x \in FV(M)$. Nevertheless, we can appropriately rename variables with careful treatment.

- Lemma 2.**
1. If $\Gamma \vdash M : \tau$, $y \notin \Gamma$ and $y \notin FV(M)$ then $\Gamma[y/x] \vdash M[y/x] : \tau$.
 2. If $\Gamma \vdash M : \tau$ and $x \notin \Gamma$ then $\Gamma, x : \sigma \vdash M : \tau$.
 3. If $\Gamma, x : \sigma \vdash M : \tau$ and $x \notin FV(M)$ then $\Gamma \vdash M : \tau$.

Proof. By induction on the structure of derivations. □

Since terms typed in an intersection type system do not in general reflect the structure of their typing derivations, a Generation Lemma is necessary for proving the subject reduction and expansion properties. Below we give a precise statement of Generation Lemma for the case of $\lambda\mathbf{x}_\cap$. To do so we first define a pre-ordering on types.

Definition 3. The relation \leq on types is defined by the following axioms and rules:

1. $\sigma \leq \sigma$
2. $\sigma \cap \tau \leq \sigma$, $\sigma \cap \tau \leq \tau$
3. $\sigma \leq \tau$, $\tau \leq \rho \Rightarrow \sigma \leq \rho$
4. $\sigma \leq \tau$, $\sigma \leq \rho \Rightarrow \sigma \leq \tau \cap \rho$

Lemma 3. If $\Gamma \vdash M : \sigma$ and $\sigma \leq \tau$ then $\Gamma \vdash M : \tau$.

Proof. By induction on the definition of \leq . □

Lemma 4. *If $\Gamma, x : \sigma \vdash M : \tau$ and $\rho \leq \sigma$ then $\Gamma, x : \rho \vdash M : \tau$.*

Proof. By induction on the derivation of $\Gamma, x : \sigma \vdash M : \tau$. \square

In the following, we use \underline{n} for $\{1, \dots, n\}$, and $\cap_{\underline{n}} \sigma_i$ for $\sigma_1 \cap \dots \cap \sigma_n$.

Lemma 5. *Let $\cap_{\underline{m}} \sigma_i \leq \cap_{\underline{n}} \tau_j$ where none of the σ_i ($i \in \underline{m}$) and τ_j ($j \in \underline{n}$) is an intersection. Then, for each τ_j , there exists σ_i such that $\sigma_i = \tau_j$.*

Proof. By induction on the definition of \leq . \square

Now we state a precise form of Generation Lemma for the system λx_{\cap} .

- Lemma 6.**
1. $\Gamma \vdash x : \tau$ if and only if there exists $x : \sigma \in \Gamma$ such that $\sigma \leq \tau$.
 2. $\Gamma \vdash MN : \tau$ if and only if there exist $\sigma_1, \dots, \sigma_n, \tau_1, \dots, \tau_n$ ($n \geq 1$) such that $\cap_{\underline{n}} \tau_i \leq \tau$ and, for all $i \in \underline{n}$, $\Gamma \vdash M : \sigma_i \rightarrow \tau_i$ and $\Gamma \vdash N : \sigma_i$.
 3. $\Gamma \vdash \lambda x.M : \tau$ if and only if there exist $\sigma_1, \dots, \sigma_n, \rho_1, \dots, \rho_n$ ($n \geq 1$) such that $\cap_{\underline{n}} (\sigma_i \rightarrow \rho_i) \leq \tau$ and, for all $i \in \underline{n}$, $\Gamma, x : \sigma_i \vdash M : \rho_i$.
 4. $\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau$ if and only if $\Gamma, x : \sigma \vdash M : \tau$.
 5. $\Gamma \vdash M \langle x := N \rangle : \tau$ if and only if either
 - (a) there exists σ such that $\Gamma \vdash N : \sigma$ and $\Gamma, x : \sigma \vdash M : \tau$, or
 - (b) $\Gamma \vdash M : \tau$ ($x \notin \Gamma$) and there exist Δ, σ such that $\Delta \vdash N : \sigma$.

Proof. The right-to-left implications immediately follow from the typing rules and Lemma 3. The converses are proved by induction on the structure of derivations, except for part 4 which follows from part 3 and Lemma 5. Here we consider the case in part 5 where the last applied rule in the derivation is $(\cap I)$:

$$\frac{\Gamma \vdash M \langle x := N \rangle : \tau_1 \quad \Gamma \vdash M \langle x := N \rangle : \tau_2}{\Gamma \vdash M \langle x := N \rangle : \tau_1 \cap \tau_2} (\cap I)$$

In this case, by the induction hypothesis, we have the following four possibilities:

- (i) there exist σ_1, σ_2 such that $\Gamma \vdash N : \sigma_1$, $\Gamma, x : \sigma_1 \vdash M : \tau_1$, $\Gamma \vdash N : \sigma_2$ and $\Gamma, x : \sigma_2 \vdash M : \tau_2$. In this case, by Lemma 4, $\Gamma, x : \sigma_1 \cap \sigma_2 \vdash M : \tau_1$ and $\Gamma, x : \sigma_1 \cap \sigma_2 \vdash M : \tau_2$, so by the rule $(\cap I)$, $\Gamma, x : \sigma_1 \cap \sigma_2 \vdash M : \tau_1 \cap \tau_2$. On the other hand, by the rule $(\cap I)$, we have $\Gamma \vdash N : \sigma_1 \cap \sigma_2$. Hence (a) holds for $\sigma = \sigma_1 \cap \sigma_2$.
- (ii) there exist $\sigma_1, \Delta, \sigma_2$ such that $\Gamma \vdash N : \sigma_1$, $\Gamma, x : \sigma_1 \vdash M : \tau_1$, $\Gamma \vdash M : \tau_2$ and $\Delta \vdash N : \sigma_2$. In this case, by Lemma 2 (2), we have $\Gamma, x : \sigma_1 \vdash M : \tau_2$, so by the rule $(\cap I)$, $\Gamma, x : \sigma_1 \vdash M : \tau_1 \cap \tau_2$. Hence (a) holds for $\sigma = \sigma_1$.
- (iii) there exist $\Delta, \sigma_1, \sigma_2$ such that $\Gamma \vdash M : \tau_1$, $\Delta \vdash N : \sigma_1$, $\Gamma \vdash N : \sigma_2$ and $\Gamma, x : \sigma_2 \vdash M : \tau_2$. This case is proved similarly to the case (ii).
- (iv) there exist $\Delta, \sigma_1, \Delta', \sigma_2$ such that $\Gamma \vdash M : \tau_1$ ($x \notin \Gamma$), $\Delta \vdash N : \sigma_1$, $\Gamma \vdash M : \tau_2$ and $\Delta' \vdash N : \sigma_2$. In this case, by the rule $(\cap I)$, we have $\Gamma \vdash M : \tau_1 \cap \tau_2$. Hence (b) holds. \square

The next lemma shows that the root reduction and expansion by the rules (App) and (Abs) preserve types.

- Lemma 7.** 1. $\Gamma \vdash (MM')\langle x := N \rangle : \tau$ if and only if $\Gamma \vdash M\langle x := N \rangle M'\langle x := N \rangle : \tau$.
2. $\Gamma \vdash (\lambda y.M)\langle x := N \rangle : \tau$ if and only if $\Gamma \vdash \lambda y.M\langle x := N \rangle : \tau$ ($y \notin FV(N)$).

Proof. Here we only show part 1.

(\Rightarrow) Let $\Gamma \vdash (MM')\langle x := N \rangle : \tau$. Then by Lemma 6 (5), we have two cases:

- (i) there exists σ such that $\Gamma \vdash N : \sigma$ and $\Gamma, x : \sigma \vdash MM' : \tau$. In this case, by Lemma 6 (2), there exist $\rho_1, \dots, \rho_n, \tau_1, \dots, \tau_n$ such that $\bigcap_{\underline{n}} \tau_i \leq \tau$ and, for all $i \in \underline{n}$, $\Gamma, x : \sigma \vdash M : \rho_i \rightarrow \tau_i$ and $\Gamma, x : \sigma \vdash M' : \rho_i$. Then by the rule (*Cut*), for each $i \in \underline{n}$, $\Gamma \vdash M\langle x := N \rangle : \rho_i \rightarrow \tau_i$ and $\Gamma \vdash M'\langle x := N \rangle : \rho_i$, so by the rule ($\rightarrow E$), $\Gamma \vdash M\langle x := N \rangle M'\langle x := N \rangle : \tau_i$. Hence by the rule ($\bigcap I$), $\Gamma \vdash M\langle x := N \rangle M'\langle x := N \rangle : \bigcap_{\underline{n}} \tau_i$, and by Lemma 3, we obtain $\Gamma \vdash M\langle x := N \rangle M'\langle x := N \rangle : \tau$.
- (ii) $\Gamma \vdash MM' : \tau$ ($x \notin \Gamma$) and there exist Δ, σ such that $\Delta \vdash N : \sigma$. This case is proved similarly to the case (i), using (*K-cut*) instead of (*Cut*).

(\Leftarrow) Let $\Gamma \vdash M\langle x := N \rangle M'\langle x := N \rangle : \tau$. Then by Lemma 6 (2), there exist $\sigma_1, \dots, \sigma_n, \tau_1, \dots, \tau_n$ such that $\bigcap_{\underline{n}} \tau_i \leq \tau$ and, for all $i \in \underline{n}$, $\Gamma \vdash M\langle x := N \rangle : \sigma_i \rightarrow \tau_i$ and $\Gamma \vdash M'\langle x := N \rangle : \sigma_i$. By Lemma 6 (5), for each $i \in \underline{n}$, there are four possibilities:

- (i) there exist ρ, ν such that $\Gamma \vdash N : \rho, \Gamma, x : \rho \vdash M : \sigma_i \rightarrow \tau_i, \Gamma \vdash N : \nu$ and $\Gamma, x : \nu \vdash M' : \sigma_i$. In this case, by Lemma 4, $\Gamma, x : \rho \cap \nu \vdash M : \sigma_i \rightarrow \tau_i$ and $\Gamma, x : \rho \cap \nu \vdash M' : \sigma_i$, so by the rule ($\rightarrow E$), $\Gamma, x : \rho \cap \nu \vdash MM' : \tau_i$. On the other hand, by the rule ($\bigcap I$), we have $\Gamma \vdash N : \rho \cap \nu$. Hence by the rule (*Cut*), we get $\Gamma \vdash (MM')\langle x := N \rangle : \tau_i$.
- (ii) there exist ρ, Δ, ν such that $\Gamma \vdash N : \rho, \Gamma, x : \rho \vdash M : \sigma_i \rightarrow \tau_i, \Gamma \vdash M' : \sigma_i$ and $\Delta \vdash N : \nu$. In this case, by Lemma 2 (2), we have $\Gamma, x : \rho \vdash M' : \sigma_i$, so by the rule ($\rightarrow E$), $\Gamma, x : \rho \vdash MM' : \tau_i$. Hence by the rule (*Cut*), we get $\Gamma \vdash (MM')\langle x := N \rangle : \tau_i$.
- (iii) there exist Δ, ν, ρ such that $\Gamma \vdash M : \sigma_i \rightarrow \tau_i, \Delta \vdash N : \nu, \Gamma \vdash N : \rho$ and $\Gamma, x : \rho \vdash M' : \sigma_i$. In this case, by Lemma 2 (2), we have $\Gamma, x : \rho \vdash M : \sigma_i \rightarrow \tau_i$, so by the rule ($\rightarrow E$), $\Gamma, x : \rho \vdash MM' : \tau_i$. Hence by the rule (*Cut*), we get $\Gamma \vdash (MM')\langle x := N \rangle : \tau_i$.
- (iv) there exist $\Delta, \nu, \Delta', \rho$ such that $\Gamma \vdash M : \sigma_i \rightarrow \tau_i$ ($x \notin \Gamma$), $\Delta \vdash N : \nu, \Gamma \vdash M' : \sigma_i$ and $\Delta' \vdash N : \rho$. In this case, by the rule ($\rightarrow E$), we have $\Gamma \vdash MM' : \tau_i$. Hence by the rule (*K-cut*), we get $\Gamma \vdash (MM')\langle x := N \rangle : \tau_i$.

Hence by the rule ($\bigcap I$), we have $\Gamma \vdash (MM')\langle x := N \rangle : \bigcap_{\underline{n}} \tau_i$, and by Lemma 3, we obtain $\Gamma \vdash (MM')\langle x := N \rangle : \tau$. \square

Now we are in a position to show that the system λx_{\cap} satisfies the subject reduction property.

Theorem 2. If $M \rightarrow_{\lambda x} N$ and $\Gamma \vdash M : \tau$ then $\Gamma \vdash N : \tau$.

Proof. By induction on the reduction relation $\rightarrow_{\lambda x}$. First we consider the cases where the reduction is at the root.

- (Beta): Let $\Gamma \vdash (\lambda x.P)Q : \tau$. Then by Lemma 6 (2), there exist $\sigma_1, \dots, \sigma_n, \tau_1, \dots, \tau_n$ such that $\bigcap_{i \in \underline{n}} \tau_i \leq \tau$ and, for all $i \in \underline{n}$, $\Gamma \vdash \lambda x.P : \sigma_i \rightarrow \tau_i$ and $\Gamma \vdash Q : \sigma_i$. By Lemma 6 (4), for each $i \in \underline{n}$, $\Gamma, x : \sigma_i \vdash P : \tau_i$, and so $\Gamma \vdash P\langle x := Q \rangle : \tau_i$ by the rule (Cut). Hence by the rule ($\cap I$) and Lemma 3, we obtain $\Gamma \vdash P\langle x := Q \rangle : \tau$.
- (App): Let $\Gamma \vdash (PQ)\langle x := R \rangle : \tau$. Then by Lemma 7 (1), we have $\Gamma \vdash P\langle x := R \rangle Q\langle x := R \rangle : \tau$.
- (Abs): Let $\Gamma \vdash (\lambda y.P)\langle x := Q \rangle : \tau$. Then by Lemma 7 (2), we have $\Gamma \vdash \lambda y.P\langle x := Q \rangle : \tau$.
- (Var): Let $\Gamma \vdash x\langle x := Q \rangle : \tau$. Then by Lemma 6 (5), either (a) there exists σ such that $\Gamma \vdash Q : \sigma$ and $\Gamma, x : \sigma \vdash x : \tau$, or (b) $\Gamma \vdash x : \tau$ ($x \notin \Gamma$) and there exist Δ, σ such that $\Delta \vdash Q : \sigma$. However, by Lemma 6 (1), (b) is impossible and we have $\sigma \leq \tau$ from $\Gamma, x : \sigma \vdash x : \tau$ in (a). Hence by Lemma 3, we obtain $\Gamma \vdash Q : \tau$.
- (gc): Let $\Gamma \vdash P\langle x := Q \rangle : \tau$ and $x \notin FV(P)$. Then by Lemma 6 (5), either (a) there exists σ such that $\Gamma \vdash Q : \sigma$ and $\Gamma, x : \sigma \vdash P : \tau$, or (b) $\Gamma \vdash P : \tau$ ($x \notin \Gamma$) and there exist Δ, σ such that $\Delta \vdash Q : \sigma$. In the case (a), we have $\Gamma \vdash P : \tau$ by Lemma 2 (3).

The cases where the reduction is not at the root are immediate by Lemma 6 and the induction hypothesis. \square

Now we can prove one direction of the characterization theorem of strongly normalizing λx -terms.

Theorem 3. *If M is typable in the system λx_{\cap} then $M \in \mathcal{SN}_{\lambda x}$.*

Proof. By Theorem 1, it suffices to show that if M is typable in the system λx_{\cap} then $M \in \lambda x^{<\infty}$. Let M be typable in λx_{\cap} and N be any subterm of M . Then N is also typable in λx_{\cap} , and by Theorem 2, so is $x(N)$. Since $x(N)$ is a pure λ -term (Proposition 2), it is typable without using (Cut) and (K -cut). Hence by the result in [19], we have $x(N) \in \mathcal{SN}_{\beta}$. Thus we obtain $M \in \lambda x^{<\infty}$. \square

In [18], a method for deriving strong normalization of typable terms from the PSN property was formalized, but it does not work for the system λx_{\cap} . The method requires preservation of typability during the lift of the explicit substitutions into β -redexes. However, the λx -term $z\langle y := xx \rangle\langle x := \lambda a.aa \rangle$ (the counter example for the system [9]) is typable in λx_{\cap} while the result of lifting ($\lambda x.(\lambda y.z)(xx)(\lambda a.aa)$) is not, so that one cannot infer strong normalization of $z\langle y := xx \rangle\langle x := \lambda a.aa \rangle$ by that method.

Next we prove the converse of Theorem 3. For this we introduce the notion of typably decent terms.

Definition 4 (Typably decent terms). *A λx -term M is said to be typably decent if for every substitution $\langle x := N \rangle$ occurring in M , N is typable in λx_{\cap} .*

Lemma 8. *If M is typably decent, $M \rightarrow_x N$ and $\Gamma \vdash N : \tau$, then $\Gamma \vdash M : \tau$.*

Proof. By induction on the reduction relation \rightarrow_x . First we consider the cases where the reduction is at the root.

- (App): Let $\Gamma \vdash P\langle x := R \rangle Q\langle x := R \rangle : \tau$. Then by Lemma 7 (1), we have $\Gamma \vdash (PQ)\langle x := R \rangle : \tau$.
- (Abs): Let $\Gamma \vdash \lambda y.P\langle x := Q \rangle : \tau$. Then by Lemma 7 (2), we have $\Gamma \vdash (\lambda y.P)\langle x := Q \rangle : \tau$.
- (Var): Let $\Gamma \vdash Q : \tau$. Since $M \equiv x\langle x := Q \rangle$ for a fresh variable x , we have $\Gamma, x : \tau \vdash x : \tau$, and so $\Gamma \vdash x\langle x := Q \rangle : \tau$ by the rule (Cut).
- (gc): Let $\Gamma \vdash P : \tau$. Since $M \equiv P\langle x := Q \rangle$ for a fresh variable x and M is typably decent, there exist Δ, σ such that $\Delta \vdash Q : \sigma$. Hence we obtain $\Gamma \vdash P\langle x := Q \rangle : \tau$ by the rule (K-cut).

The cases where the reduction is not at the root are immediate by Lemma 6 and the induction hypothesis. □

Lemma 9. *If M is typably decent and $M \rightarrow_x N$, then N is typably decent.*

Proof. By induction on the reduction relation \rightarrow_x . The cases where the reduction is at the root are straightforward, since every substitution body occurring in N also occurs in M . Let us consider the case $M \equiv P\langle x := Q \rangle$ and $Q \rightarrow_x Q'$. Since M is typably decent, Q is typable in $\lambda\mathbf{x}_\cap$, so by Theorem 2, Q' is typable in $\lambda\mathbf{x}_\cap$. Hence we see that $P\langle x := Q' \rangle$ is typably decent. □

Lemma 10. *If M is typably decent, $M \overset{*}{\rightarrow}_x N$ and $\Gamma \vdash N : \tau$, then $\Gamma \vdash M : \tau$.*

Proof. By induction on the length of the reduction steps of $M \overset{*}{\rightarrow}_x N$, using Lemmas 8 and 9. □

Now we can prove the converse of Theorem 3.

Theorem 4. *If $M \in \mathcal{SN}_{\lambda\mathbf{x}}$ then M is typable in the system $\lambda\mathbf{x}_\cap$.*

Proof. By induction on the structure of M . Suppose that $M \in \mathcal{SN}_{\lambda\mathbf{x}}$. Then for every substitution $\langle x := N \rangle$ occurring in M , $N \in \mathcal{SN}_{\lambda\mathbf{x}}$, so by the induction hypothesis, N is typable in $\lambda\mathbf{x}_\cap$. Hence M is typably decent. On the other hand, since $M \in \mathcal{SN}_{\lambda\mathbf{x}}$, we have $\mathbf{x}(M) \in \mathcal{SN}_{\lambda\mathbf{x}}$, so by Lemma 1, $\mathbf{x}(M) \in \mathcal{SN}_\beta$. Hence by the result in 19, $\mathbf{x}(M)$ is typable in $\lambda\mathbf{x}_\cap$ (without using (Cut) and (K-cut)). Therefore by Lemma 10, M is typable in $\lambda\mathbf{x}_\cap$. □

4 $\overline{\lambda\mathbf{x}}$ -Calculus

In the remainder of the paper we study an explicit substitution calculus which we call here $\overline{\lambda\mathbf{x}}$ -calculus. This calculus is to sequent calculus what $\lambda\mathbf{x}$ -calculus is to natural deduction. Simply typed terms of the calculus correspond to proofs in Herbelin’s sequent calculus 12, which has a stronger connection with λ -calculus than the usual sequent calculus does; in particular, it relates a unique cut-free proof to each normal term of the simply typed λ -calculus. The reduction rules of Herbelin’s original calculus were later extended by Dyckhoff and Urban 11 to simulate full β -reduction. In the simply typed case, they correspond to cut-elimination steps in the sequent calculus.

Table 3. $\bar{\lambda}\mathbf{x}$ -calculus

$t, u, v ::= xl \mid \lambda x.t \mid tl \mid t\langle x := v \rangle$ $l, l' ::= [] \mid t :: l \mid ll' \mid l\langle x := v \rangle$	
(Beta)	$(\lambda x.t)(u :: l) \rightarrow t\langle x := u \rangle l$
(1a)	$[]l \rightarrow l$
(1b)	$(u :: l)l' \rightarrow u :: (ll')$
(2a)	$[]\langle x := v \rangle \rightarrow []$
(2b)	$(u :: l)\langle x := v \rangle \rightarrow u\langle x := v \rangle :: l\langle x := v \rangle$
(3a)	$(xl)l' \rightarrow x(ll')$
(3b)	$(\lambda y.t)[] \rightarrow \lambda y.t$
(4a)	$(yl)\langle x := v \rangle \rightarrow yl\langle x := v \rangle \quad \text{if } y \neq x$
(4b)	$(xl)\langle x := v \rangle \rightarrow vl\langle x := v \rangle$
(4c)	$(\lambda y.t)\langle x := v \rangle \rightarrow \lambda y.t\langle x := v \rangle$
(5a)	$(ll')l'' \rightarrow l(l'l'')$
(5b)	$(ll')\langle x := v \rangle \rightarrow l\langle x := v \rangle l'\langle x := v \rangle$
(5c)	$(tl)l' \rightarrow t(ll')$
(5d)	$(tl)\langle x := v \rangle \rightarrow t\langle x := v \rangle l\langle x := v \rangle$

Table 3 gives the syntax and the reduction rules of $\bar{\lambda}\mathbf{x}$ -calculus. The syntax has two kinds of expressions: terms and lists of terms, ranged over by t, u, v and by l, l' , respectively. The set of terms is denoted by $\mathcal{T}_{\bar{\lambda}\mathbf{x}}$ and the set of lists of terms by $\mathcal{L}_{\bar{\lambda}\mathbf{x}}$. Elements of $\mathcal{T}_{\bar{\lambda}\mathbf{x}} \cup \mathcal{L}_{\bar{\lambda}\mathbf{x}}$ are called $\bar{\lambda}\mathbf{x}$ -terms and ranged over by a, b . The notions of free and bound variables are defined as in the case of $\lambda\mathbf{x}$ -terms.

To see the relation to ordinary λ -calculus, it is useful to consider a subset of $\bar{\lambda}\mathbf{x}$ -terms defined by the following grammar:

$$t, u, v ::= xl \mid \lambda x.t \mid (\lambda x.t)(u :: l)$$

$$l, l' ::= [] \mid t :: l$$

The $\bar{\lambda}\mathbf{x}$ -terms generated by this grammar are called *pure terms*. Then compare the grammar of pure terms with the following inductive characterization of the set of pure λ -terms:

$$M, N ::= xM_1 \dots M_n \mid \lambda x.M \mid (\lambda x.M)NM_1 \dots M_n \quad (n \geq 0)$$

Note that this certainly generates all pure λ -terms. Now it is easy to see that there exists one-to-one correspondence between pure λ -terms and pure terms in $\mathcal{T}_{\bar{\lambda}\mathbf{x}}$. We denote the bijection from pure λ -terms to pure terms by Ψ . Moreover we define β -reduction on pure terms in $\mathcal{T}_{\bar{\lambda}\mathbf{x}}$ so that it coincides with β -reduction on

pure λ -terms under the bijection, i.e., for any pure λ -terms $M, M', M \rightarrow_\beta M'$ if and only if $\Psi(M) \rightarrow_\beta \Psi(M')$. (The β -reduction extends to pure terms in $\mathcal{L}_{\bar{\lambda}\mathbf{x}}$.)

The notion of $\bar{\lambda}\mathbf{x}$ -reduction is defined by the contextual closures of all reduction rules in Table 3. Then $\bar{\lambda}\mathbf{x}$ -calculus works as an explicit substitution calculus for the isomorphic image of λ -calculus. The reduction properties of $\bar{\lambda}\mathbf{x}$ -calculus are similar to those of $\lambda\mathbf{x}$ -calculus. In the following we summarize results on the subcalculus \mathbf{x} (i.e., the calculus without the rule (*Beta*)) and the relation between $\bar{\lambda}\mathbf{x}$ -reduction and β -reduction on pure terms. (For details, see [11,15].)

Proposition 4. *The subcalculus \mathbf{x} is strongly normalizing and confluent.*

Definition 5. *The unique \mathbf{x} -normal form of a $\bar{\lambda}\mathbf{x}$ -term a is denoted by $\mathbf{x}(a)$.*

Proposition 5. *a is a pure term if and only if a is in \mathbf{x} -normal form.*

Lemma 11. *For any pure terms $a, b \in \mathcal{T}_{\bar{\lambda}\mathbf{x}} \cup \mathcal{L}_{\bar{\lambda}\mathbf{x}}$, if $a \rightarrow_\beta b$ then $a \xrightarrow{+}_{\bar{\lambda}\mathbf{x}} b$.*

Using a similar technique to the one in [4], Dyckhoff and Urban [11] proved the following theorem. We use this theorem to characterize strongly normalizing $\bar{\lambda}\mathbf{x}$ -terms by an intersection type assignment system in the next section.

Definition 6 (Bounded terms). *The set of bounded terms, denoted $\bar{\lambda}\mathbf{x}^{<\infty}$, is defined by $\bar{\lambda}\mathbf{x}^{<\infty} =_{\text{def}} \{a \mid \text{for every subterm } b \text{ of } a, \mathbf{x}(b) \in \mathcal{SN}_\beta\}$.*

Theorem 5 ([11]). *If $a \in \bar{\lambda}\mathbf{x}^{<\infty}$ then $a \in \mathcal{SN}_{\bar{\lambda}\mathbf{x}}$.*

5 Characterization of Strongly Normalizing $\bar{\lambda}\mathbf{x}$ -Terms

In this section we introduce a new intersection type assignment system in the style of sequent calculus. The system is an extension of Herbelin’s type assignment system with simple types in [12]. We show that the strongly normalizing $\bar{\lambda}\mathbf{x}$ -terms coincide with those typable in the intersection type assignment system in a similar way to that in Section 3.

First we extend the pre-ordering \leq to one in the style of [2], which reduces the difficulty of proving the subject reduction property of our intersection type assignment system.

Definition 7. *The relation \leq on types is defined by the axioms and rules 1–4 in Definition 3 and the following:*

$$5. (\sigma \rightarrow \tau) \cap (\sigma \rightarrow \rho) \leq \sigma \rightarrow (\tau \cap \rho) \quad 6. \sigma' \leq \sigma, \tau \leq \tau' \Rightarrow \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'$$

Lemma 12. $(\sigma_1 \rightarrow \tau_1) \cap (\sigma_2 \rightarrow \tau_2) \leq (\sigma_1 \cap \sigma_2) \rightarrow (\tau_1 \cap \tau_2)$.

Lemma 13. *If $\bigcap_{i \in \underline{n}} (\mu_i \rightarrow \nu_i) \leq \sigma \rightarrow \tau$ then there exist $i_1, \dots, i_k \in \underline{n}$ such that $\sigma \leq \mu_{i_1} \cap \dots \cap \mu_{i_k}$ and $\nu_{i_1} \cap \dots \cap \nu_{i_k} \leq \tau$.*

Proof. See Lemma 2.4 (ii) of [2]. □

Table 4. The type assignment system $\bar{\lambda}_{\mathbf{x}\cap}$

$\frac{}{\Gamma; \sigma \vdash [] : \sigma} (Ax)$	$\frac{\Gamma, x : \sigma; \sigma \vdash l : \tau}{\Gamma, x : \sigma; - \vdash xl : \tau} (Der)$	$\frac{\Gamma, x : \sigma; - \vdash t : \tau}{\Gamma; - \vdash \lambda x.t : \sigma \rightarrow \tau} (R \rightarrow)$
$\frac{\Gamma; - \vdash t : \sigma \quad \Gamma; \tau \vdash l : \rho}{\Gamma; \sigma \rightarrow \tau \vdash t :: l : \rho} (L \rightarrow)$	$\frac{\Gamma; \Pi \vdash a : \sigma \quad \Gamma; \Pi \vdash a : \tau}{\Gamma; \Pi \vdash a : \sigma \cap \tau} (R \cap)$	
$\frac{\Gamma; \sigma \vdash l : \tau \quad \rho \leq \sigma}{\Gamma; \rho \vdash l : \tau} (L \leq)$	$\frac{\Gamma; \Pi \vdash a : \sigma \quad \sigma \leq \tau}{\Gamma; \Pi \vdash a : \tau} (R \leq)$	
$\frac{\Gamma; \Pi \vdash a : \sigma \quad \Gamma; \sigma \vdash l : \tau}{\Gamma; \Pi \vdash al : \tau} (Cut_1)$	$\frac{\Gamma; - \vdash v : \sigma \quad \Gamma, x : \sigma; \Pi \vdash a : \tau}{\Gamma; \Pi \vdash a \langle x := v \rangle : \tau} (Cut_2)$	
$\frac{\Delta; - \vdash v : \sigma \quad \Gamma; \Pi \vdash a : \tau}{\Gamma; \Pi \vdash a \langle x := v \rangle : \tau} (K-cut)$		
where $x \notin \Gamma$		

Table 4 presents the rules of the type assignment system $\bar{\lambda}_{\mathbf{x}\cap}$, which is based on two kinds of judgements: $\Gamma; - \vdash t : \tau$ and $\Gamma; \sigma \vdash l : \tau$. We use $\Gamma; \Pi \vdash a : \tau$ to denote both kinds of judgements, with Π being zero or one type. The type σ in $\Gamma; \sigma \vdash l : \tau$ represents the type of a head variable to be attached to the list l . In other words, σ is the type of the hole of l , since l can be viewed as a context with a hole in the position of the head variable (cf. the comparison between pure terms and pure λ -terms in the previous section). So in the rule $(L \rightarrow)$, the hole with type τ in the right premiss is replaced, in the conclusion, by the hole with type $\sigma \rightarrow \tau$ applied to the term t which is typed with σ in the left premiss. In the rule (Cut_1) , we use the notation al , which is read as the $\bar{\lambda}_{\mathbf{x}\cap}$ -term obtained by filling the hole of l with a term or another context a .

In the following we show some lemmas on properties of the system $\bar{\lambda}_{\mathbf{x}\cap}$.

- Lemma 14.**
1. If $\Gamma; \Pi \vdash a : \tau$, $y \notin \Gamma$ and $y \notin FV(a)$ then $\Gamma[y/x]; \Pi \vdash a[y/x] : \tau$.
 2. If $\Gamma; \Pi \vdash a : \tau$ and $x \notin \Gamma$ then $\Gamma, x : \sigma; \Pi \vdash a : \tau$.
 3. If $\Gamma, x : \sigma; \Pi \vdash a : \tau$ and $x \notin FV(a)$ then $\Gamma; \Pi \vdash a : \tau$.

Proof. By induction on the structure of derivations. □

Lemma 15. If $\Gamma; \sigma \vdash [] : \tau$ then $\sigma \leq \tau$.

Proof. By induction on the derivation of $\Gamma; \sigma \vdash [] : \tau$. □

Lemma 16. If $\Gamma, x : \sigma; \Pi \vdash a : \tau$ and $\rho \leq \sigma$ then $\Gamma, x : \rho; \Pi \vdash a : \tau$.

Proof. By induction on the derivation of $\Gamma, x : \sigma; \Pi \vdash a : \tau$. \square

Lemma 17. *If $\Gamma; - \vdash xl : \tau$ then there exists σ such that $x : \sigma \in \Gamma$.*

Proof. By induction on the derivation of $\Gamma; - \vdash xl : \tau$. \square

Now we state a precise form of Generation Lemma for the system $\overline{\lambda x}_\cap$.

Lemma 18. 1. $\Gamma, x : \sigma; - \vdash xl : \tau$ if and only if $\Gamma, x : \sigma; \sigma \vdash l : \tau$.

2. $\Gamma; - \vdash \lambda x.t : \tau$ if and only if there exist $\sigma_1, \dots, \sigma_n, \rho_1, \dots, \rho_n$ ($n \geq 1$) such that $\cap_{\underline{n}}(\sigma_i \rightarrow \rho_i) \leq \tau$ and, for all $i \in \underline{n}$, $\Gamma, x : \sigma_i; - \vdash t : \rho_i$.

3. $\Gamma; - \vdash tl : \tau$ if and only if there is σ such that $\Gamma; - \vdash t : \sigma$ and $\Gamma; \sigma \vdash l : \tau$.

4. $\Gamma; - \vdash t(x := v) : \tau$ if and only if either

(a) there exists σ such that $\Gamma; - \vdash v : \sigma$ and $\Gamma, x : \sigma; - \vdash t : \tau$, or

(b) $\Gamma; - \vdash t : \tau$ ($x \notin \Gamma$) and there exist Δ, σ such that $\Delta; - \vdash v : \sigma$.

5. $\Gamma; \rho \vdash t :: l : \tau$ if and only if there exist σ, ν such that $\rho \leq \sigma \rightarrow \nu$, $\Gamma; - \vdash t : \sigma$ and $\Gamma; \nu \vdash l : \tau$.

6. $\Gamma; \rho \vdash ll' : \tau$ if and only if there is σ such that $\Gamma; \rho \vdash l : \sigma$ and $\Gamma; \sigma \vdash l' : \tau$.

7. $\Gamma; \rho \vdash l \langle x := v \rangle : \tau$ if and only if either

(a) there exists σ such that $\Gamma; - \vdash v : \sigma$ and $\Gamma, x : \sigma; \rho \vdash l : \tau$, or

(b) $\Gamma; \rho \vdash l : \tau$ ($x \notin \Gamma$) and there exist Δ, σ such that $\Delta; - \vdash v : \sigma$.

We are now in a position to show that the system $\overline{\lambda x}_\cap$ satisfies the subject reduction property.

Theorem 6. *If $a \rightarrow_{\overline{\lambda x}} b$ and $\Gamma; \Pi \vdash a : \tau$ then $\Gamma; \Pi \vdash b : \tau$.*

Finally we need the following proposition which states that the bijection Ψ preserves the typability of pure λ -terms and pure terms in $\overline{\mathcal{T}}_{\overline{\lambda x}}$.

Proposition 6. *For any pure λ -term M , M is typable in λx_\cap (without using (Cut) and (K-cut)) if and only if $\Psi(M)$ is typable in $\overline{\lambda x}_\cap$.*

Now we can prove one direction of the characterization theorem of strongly normalizing $\overline{\lambda x}$ -terms.

Theorem 7. *If a is typable in the system $\overline{\lambda x}_\cap$ then $a \in \mathcal{SN}_{\overline{\lambda x}}$.*

Proof. By Theorem 5, it suffices to show that if a is typable in the system $\overline{\lambda x}_\cap$ then $a \in \overline{\lambda x}^{<\infty}$. Let a be typable in $\overline{\lambda x}_\cap$ and b be any subterm of a . Then b is also typable in $\overline{\lambda x}_\cap$, and by Theorem 6, so is $x(b)$. If $x(b) \in \overline{\mathcal{T}}_{\overline{\lambda x}}$ then by Proposition 6, $\Psi^{-1}(x(b))$ is typable, and so $\Psi^{-1}(x(b)) \in \mathcal{SN}_\beta$ by the result in 19. Hence by the definition of β -reduction on pure terms, we have $x(b) \in \mathcal{SN}_\beta$. If $x(b) \in \mathcal{L}_{\overline{\lambda x}}$ then each element $u \in \overline{\mathcal{T}}_{\overline{\lambda x}}$ of the list $x(b)$ must be reduced independently, and so $x(b) \in \mathcal{SN}_\beta$. Thus we obtain $a \in \overline{\lambda x}^{<\infty}$. \square

The above theorem extends the results of 11,15 where strong normalization is proved for $\overline{\lambda x}$ -terms typed with simple types. Our system $\overline{\lambda x}_\cap$ is, in fact, able to type all strongly normalizing $\overline{\lambda x}$ -terms. To show that, we introduce again the notion of typably decent terms.

Definition 8 (Typably decent terms). A $\bar{\lambda}\mathbf{x}$ -term a is said to be typably decent if for every substitution $\langle x := v \rangle$ occurring in a , v is typable in $\bar{\lambda}\mathbf{x}_\Gamma$.

Lemma 19. If a is typably decent, $a \rightarrow_{\mathbf{x}} b$ and $\Gamma; \Pi \vdash b : \tau$, then $\Gamma; \Pi \vdash a : \tau$.

Lemma 20. If a is typably decent and $a \rightarrow_{\mathbf{x}} b$, then b is typably decent.

Proof. Similar to the proof of Lemma 9. □

Lemma 21. If a is typably decent, $a \xrightarrow{*}_{\mathbf{x}} b$ and $\Gamma; \Pi \vdash b : \tau$, then $\Gamma; \Pi \vdash a : \tau$.

Proof. By induction on the length of the reduction steps of $a \xrightarrow{*}_{\mathbf{x}} b$, using Lemmas 19 and 20. □

Now we can prove the converse of Theorem 7.

Theorem 8. If $a \in \mathcal{SN}_{\bar{\lambda}\mathbf{x}}$ then a is typable in the system $\bar{\lambda}\mathbf{x}_\Gamma$.

Proof. By induction on the structure of a . Suppose that $a \in \mathcal{SN}_{\bar{\lambda}\mathbf{x}}$. Then for every substitution $\langle x := v \rangle$ occurring in a , $v \in \mathcal{SN}_{\bar{\lambda}\mathbf{x}}$, so by the induction hypothesis, v is typable in $\bar{\lambda}\mathbf{x}_\Gamma$. Hence a is typably decent. On the other hand, since $a \in \mathcal{SN}_{\bar{\lambda}\mathbf{x}}$, we have $\mathbf{x}(a) \in \mathcal{SN}_{\bar{\lambda}\mathbf{x}}$, so by Lemma 11, $\mathbf{x}(a) \in \mathcal{SN}_\beta$.

If $\mathbf{x}(a) \in \mathcal{T}_{\bar{\lambda}\mathbf{x}}$ then by the definition of β -reduction on pure terms, we have $\Psi^{-1}(\mathbf{x}(a)) \in \mathcal{SN}_\beta$, so by the result in 19, $\Psi^{-1}(\mathbf{x}(a))$ is typable in $\lambda\mathbf{x}_\Gamma$ (without using *(Cut)* and *(K-cut)*). Hence by Proposition 6, $\mathbf{x}(a)$ is typable in $\bar{\lambda}\mathbf{x}_\Gamma$. Therefore by Lemma 21, a is typable in $\bar{\lambda}\mathbf{x}_\Gamma$.

If $\mathbf{x}(a) \in \mathcal{L}_{\bar{\lambda}\mathbf{x}}$ then for any variable y , $y\mathbf{x}(a)$ is a pure term in $\mathcal{T}_{\bar{\lambda}\mathbf{x}}$. Since $\mathbf{x}(a) \in \mathcal{SN}_\beta$, we have $y\mathbf{x}(a) \in \mathcal{SN}_\beta$, so by the above argument, $y\mathbf{x}(a)$ is typable in $\bar{\lambda}\mathbf{x}_\Gamma$. Hence $\mathbf{x}(a)$ is typable in $\bar{\lambda}\mathbf{x}_\Gamma$, and by Lemma 21, a is typable in $\bar{\lambda}\mathbf{x}_\Gamma$. □

6 Conclusion

In this paper, we presented a method for lifting to explicit substitution calculi characterizations of the strongly normalizing terms of λ -calculus by means of intersection type systems. In the first half of the paper, we gave a simple proof of characterizing the strongly normalizing terms of $\lambda\mathbf{x}$ -calculus by an intersection type system in 16. In the latter half of the paper, we characterized the strongly normalizing terms of the explicit substitution calculus of 11 by a new intersection type system based on sequent calculus.

A challenging problem is to characterize the strongly normalizing terms of $\bar{\lambda}\mu\tilde{\mu}$ -calculus 7 with explicit substitutions explored in 18. For $\bar{\lambda}\mu\tilde{\mu}$ -calculus without explicit substitutions, Dougherty et al. 10 studied characterization of the strongly normalizing terms, using intersection and union types. For having the subject reduction property, they imposed some restrictions on types of variables, whereas we introduced subtyping rules with a (semantically justified) pre-ordering. Another direction for further work is to extend our technique to explicit substitution calculi with composition and/or equations like the calculus in 13.

Acknowledgements. I am grateful to Takafumi Sakurai for useful discussions and Stéphane Lengrand for his interest in this work. I also thank Roy Dyckhoff and the anonymous referees for valuable comments. This research was partially supported by the Japanese Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) 17700003.

References

1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. *J. Funct. Program.* 1, 375–416 (1991)
2. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A filter lambda model and the completeness of type assignment. *J. Symb. Log.* 48, 931–940 (1983)
3. Bloo, R.: Preservation of Termination for Explicit Substitution. PhD thesis, Eindhoven University of Technology (1997)
4. Bloo, R., Geuvers, H.: Explicit substitution: On the edge of strong normalization. *Theor. Comput. Sci.* 211, 375–395 (1999)
5. Bloo, R., Rose, K.H.: Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In: Proceedings of CSN’95 (Computing Science in the Netherlands), pp. 62–72 (1995)
6. Bonelli, E.: Perpetuality in a named lambda calculus with explicit substitutions. *Math. Structures Comput. Sci.* 11, 47–90 (2001)
7. Curien, P.-L., Herbelin, H.: The duality of computation. In: Proceedings of ICFP’00, pp. 233–243 (2000)
8. Dershowitz, N.: Orderings for term-rewriting systems. *Theor. Comput. Sci.* 17, 279–301 (1982)
9. Dougherty, D., Lescanne, P.: Reductions, intersection types, and explicit substitutions. *Math. Structures Comput. Sci.* 13, 55–85 (2003)
10. Dougherty, D., Ghilezan, S., Lescanne, P.: Characterizing strong normalization in a language with control operators. In: Proceedings of PPDP’04, pp. 155–166 (2004)
11. Dyckhoff, R., Urban, C.: Strong normalization of Herbelin’s explicit substitution calculus with substitution propagation. *J. Log. Comput.* 13, 689–706 (2003)
12. Herbelin, H.: A λ -calculus structure isomorphic to Gentzen-style sequent calculus structure. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 61–75. Springer, Heidelberg (1995)
13. Kesner, D., Lengrand, S.: Resource operators for the λ -calculus. *Inform. and Comput.* 205, 419–473 (2007)
14. Khasidashvili, Z., Ogawa, M., van Oostrom, V.: Uniform normalisation beyond orthogonality. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 122–136. Springer, Heidelberg (2001)
15. Kikuchi, K.: A direct proof of strong normalization for an extended Herbelin’s calculus. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 244–259. Springer, Heidelberg (2004)
16. Lengrand, S., Lescanne, P., Dougherty, D., Dezani-Ciancaglini, M., van Bakel, S.: Intersection types for explicit substitutions. *Inform. and Comput.* 189, 17–42 (2004)
17. Mellès, P.-A.: Typed λ -calculi with explicit substitutions may not terminate. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 328–334. Springer, Heidelberg (1995)

18. Polonovski, E.: Strong normalization of $\overline{\lambda\mu\tilde{\iota}}$ -calculus with explicit substitutions. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 423–437. Springer, Heidelberg (2004)
19. Pottinger, G.: A type assignment for the strongly normalizable λ -terms. In: To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 561–577. Academic Press, San Diego (1980)
20. van Raamsdonk, F., Severi, P.: On normalisation. Technical Report CS-R9545, CWI (1995)

Proving Termination of Rewrite Systems Using Bounds

Martin Korp and Aart Middeldorp

Institute of Computer Science
University of Innsbruck
Austria

Abstract. The use of automata techniques to prove the termination of string rewrite systems and left-linear term rewrite systems is advocated by Geser *et al.* in a recent sequence of papers. We extend their work to non-left-linear rewrite systems. The key to this extension is the introduction of so-called raise rules and the use of tree automata that are not quite deterministic. Furthermore, we present negative solutions to two open problems related to string rewrite systems.

1 Introduction

Using automata techniques is a relatively new and elegant approach for automatically proving the termination of rewrite systems. Initially proposed for string rewriting by Geser, Hofbauer, and Waldmann [7], the method has recently been extended to left-linear term rewrite systems [10]. Variations and improvements are discussed in [5,8,9]. The fact that the method has been implemented in several different termination provers ([4,12,16,17]) is a clear witness of the success of the approach.

In this paper we look at two extremes. On the one hand, we present a negative solution to the problem whether a given string rewrite system can be proved terminating by the method if no a priori bound is given. A simple reduction from the undecidable termination problem for string rewrite systems does the trick. We further show that failure of the method is not completely characterized by the presence of a so-called witnessing set. Both results settle open problems in [7].

On the other hand, we extend the method to term rewrite systems containing rules that are not left-linear. This turns out to be surprisingly challenging. First of all, the theory on which the method is based does not work without further ado for non-left-linear rewrite systems. So-called raise rules are introduced to solve this issue. Second, the usual approach of using deterministic tree automata for dealing with non-left-linear rewrite rules appears to be incompatible with the method. We introduce quasi-deterministic tree automata to overcome this problem. Finally, the raise rules need special care to enable the automata construction to terminate.

The remainder of the paper is organized as follows. In the next section we recall basic definitions concerning the automata theory approach to proving termination of rewrite systems. In Section 3 we introduce raise rules to overcome the

problem caused by non-left-linear rules. Quasi-deterministic tree automata are introduced in Section 4. In Sections 5 and 6 it is explained how these automata are used to infer termination. Like in the linear case, the power of the method is increased by considering right-hand sides of forward closures. This is explained in Section 7. We present experimental data in Section 8. In Section 9 we present our negative solutions to the open problems for string rewrite systems.

2 Preliminaries

We assume familiarity with term rewriting [1] and tree automata [2]. General knowledge of the match-bound technique [7,10] will be helpful. Below we recall important definitions and results from the latter paper.

A TRS \mathcal{R} over a signature \mathcal{F} is called *locally terminating* if every restriction of \mathcal{R} to a finite signature $\mathcal{G} \subseteq \mathcal{F}$ is terminating. Given a set $L \subseteq \mathcal{T}(\mathcal{F})$ of ground terms, we denote the set $\{t \in \mathcal{T}(\mathcal{F}) \mid s \rightarrow_{\mathcal{R}}^* t \text{ for some } s \in L\}$ of descendants of L by $\rightarrow_{\mathcal{R}}^*(L)$. Given a set $N \subseteq \mathbb{N}$ of natural numbers, the signature $\mathcal{F} \times N$ is denoted by \mathcal{F}_N . Here function symbols (f, n) with $f \in \mathcal{F}$ and $n \in N$ have the same arity as f and are written as f_n . Let \mathcal{F} be a signature. The mappings $\text{lift}_c: \mathcal{F} \rightarrow \mathcal{F}_N$, $\text{base}: \mathcal{F}_N \rightarrow \mathcal{F}$, and $\text{height}: \mathcal{F}_N \rightarrow \mathbb{N}$ are defined as follows:

$$\text{lift}_c(f) = f_c \quad \text{base}(f_i) = f \quad \text{height}(f_i) = i$$

for all $f \in \mathcal{F}$ and $c, i \in \mathbb{N}$. They are extended to terms and to set of terms in the obvious way.

Let t be a term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and $V \subseteq \text{Var}(t)$ a set of variables. A position $p \in \mathcal{F}\text{Pos}(t)$ is a *roof position* in t for V if $V \subseteq \text{Var}(t|_p)$. The set of all roof positions in t for V is denoted by $\mathcal{R}\text{Pos}_V(t)$. Let l and r be two terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The mappings top , roof , and match are defined as follows:

$$\text{top}(l, r) = \{\epsilon\} \quad \text{roof}(l, r) = \mathcal{R}\text{Pos}_{\text{Var}(r)}(l) \quad \text{match}(l, r) = \mathcal{F}\text{Pos}(l)$$

Let \mathcal{R} be a TRS over the signature \mathcal{F} and e a function that maps every rewrite rule $l \rightarrow r \in \mathcal{R}$ to a nonempty subset of $\mathcal{F}\text{Pos}(l)$. The TRS $e(\mathcal{R})$ over the signature \mathcal{F}_N consists of all rewrite rules $l' \rightarrow \text{lift}_c(r)$ for which there exists a rule $l \rightarrow r \in \mathcal{R}$ such that $\text{base}(l') = l$ and $c = 1 + \min\{\text{height}(l'(p)) \mid p \in e(l, r)\}$. Let $c \in \mathbb{N}$. The restriction of $e(\mathcal{R})$ to the signature $\mathcal{F}_{\{0, \dots, c\}}$ is denoted by $e_c(\mathcal{R})$. Let $e \in \{\text{top}, \text{roof}, \text{match}\}$ and L a set of terms. The TRS \mathcal{R} is called *e-bounded* for L if there exists a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms in $\rightarrow_{e_c(\mathcal{R})}^*(\text{lift}_0(L))$ is at most c . If we want to precise the bound c , we say that \mathcal{R} is *e-bounded for L by c* . In the following we do not mention L if we have the set of all ground terms in mind.

Lemma 1 ([10]). *Let \mathcal{R} be a TRS. The TRSs $\text{top}(\mathcal{R})$ and $\text{roof}(\mathcal{R})$ are locally terminating. If \mathcal{R} is right-linear then $\text{match}(\mathcal{R})$ is locally terminating. \square*

3 Raise-Bounds

The following example shows that e -bounded TRSs need not be terminating.

Example 2. Consider the non-terminating TRS $\mathcal{R} = \{f(x, x) \rightarrow f(a, x)\}$. The TRSs $\text{match}(\mathcal{R})$, $\text{roof}(\mathcal{R})$, and $\text{top}(\mathcal{R})$ coincide and consist of the rules

$$f_i(x, x) \rightarrow f_{i+1}(a_{i+1}, x)$$

for all $i \geq 0$. It is not difficult to see that with these rules we can never reach height 2 starting from a term in $\mathcal{T}(\{a_0, f_0\})$. Hence \mathcal{R} is e -bounded by 1 for all $e \in \{\text{top}, \text{roof}, \text{match}\}$.

The problem is that even though every single \mathcal{R} -step can be simulated by an $e(\mathcal{R})$ -step, this does not hold for consecutive \mathcal{R} -steps. We have $f(a, a) \rightarrow_{\mathcal{R}} f(a, a) \rightarrow_{\mathcal{R}} f(a, a)$ but after the step $f_0(a_0, a_0) \rightarrow_{e(\mathcal{R})} f_1(a_1, a_0)$ we are stuck because $a_0 \neq a_1$.

Definition 3. Let \mathcal{F} be a signature. The TRS $\text{raise}(\mathcal{F})$ over the signature $\mathcal{F}_{\mathbb{N}}$ consists of all rules

$$f_i(x_1, \dots, x_n) \rightarrow f_{i+1}(x_1, \dots, x_n)$$

with f an n -ary function symbol in \mathcal{F} , $i \in \mathbb{N}$, and x_1, \dots, x_n pairwise different variables. The restriction of $\text{raise}(\mathcal{F})$ to the signature $\mathcal{F}_{\{0, \dots, c\}}$ is denoted by $\text{raise}_c(\mathcal{F})$. For terms $s, t \in \mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$ we write $s \leq t$ if $s \rightarrow_{\text{raise}(\mathcal{F})}^* t$ and $s \uparrow t$ for the least term u with $s \leq u$ and $t \leq u$. The latter notion is extended to $\uparrow S$ for finite nonempty sets $S \subset \mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$ in the obvious way.

The following result corresponds to Lemma [11](#). The right-linearity condition is weakened to non-duplication in order to cover more non-left-linear TRSs. (A TRS is duplicating if there exist a rewrite rule $l \rightarrow r$ and a variable x that occurs more often in r than in l .)

Lemma 4. Let \mathcal{R} be a TRS over a signature \mathcal{F} . The TRSs $\text{top}(\mathcal{R}) \cup \text{raise}(\mathcal{F})$ and $\text{roof}(\mathcal{R}) \cup \text{raise}(\mathcal{F})$ are locally terminating. If \mathcal{R} is non-duplicating then $\text{match}(\mathcal{R}) \cup \text{raise}(\mathcal{F})$ is locally terminating.

Proof. Straightforward adaptations of the proofs of Lemmata [16](#) and [17](#) in [\[10\]](#). □

An immediate consequence of the next lemma states that every derivation in \mathcal{R} can be simulated using the rules in $e(\mathcal{R})$ and $\text{raise}(\mathcal{F})$.

Lemma 5. Let \mathcal{R} be a TRS over a signature \mathcal{F} . If $s \rightarrow_{\mathcal{R}} t$ then for all s' with $\text{base}(s') = s$ there exists a term t' such that $\text{base}(t') = t$ and $s' \rightarrow_{e(\mathcal{R}) \cup \text{raise}(\mathcal{F})}^+ t'$.

Proof. Straightforward. □

However, since $\text{raise}(\mathcal{F})$ is non-terminating, in order to use $e(\mathcal{R}) \cup \text{raise}(\mathcal{F})$ to infer termination of \mathcal{R} , we have to restrict the rules of $\text{raise}(\mathcal{F})$ to those that

are really needed to simulate derivations in \mathcal{R} . We do this by defining a new relation $\overset{\geq}{\rightarrow}_{e(\mathcal{R})}$ in which the necessary raise steps are built in. The idea is that $s \overset{\geq}{\rightarrow}_{e(\mathcal{R})} t$ if t can be obtained from s by doing the minimum number of raise steps to ensure the applicability of a non-left-linear rewrite rule in $e(\mathcal{R})$.

Definition 6. Let \mathcal{R} be a TRS over a signature \mathcal{F} . We define the relation $\overset{\geq}{\rightarrow}_{e(\mathcal{R})}$ on $\mathcal{T}(\mathcal{F}_{\mathbb{N}}, \mathcal{V})$ as follows: $s \overset{\geq}{\rightarrow}_{e(\mathcal{R})} t$ if and only if there exist a rewrite rule $l \rightarrow r \in e(\mathcal{R})$, a position $p \in \text{Pos}(s)$, a context C , and terms s_1, \dots, s_n such that $l = C[x_1, \dots, x_n]$ with all variables displayed, $s|_p = C[s_1, \dots, s_n]$, $\text{base}(s_i) = \text{base}(s_j)$ whenever $x_i = x_j$, and $t = s[r\theta]_p$. Here the substitution θ is defined as follows:

$$\theta(x) = \begin{cases} \uparrow \{s_i \mid x_i = x\} & \text{if } x \in \{x_1, \dots, x_n\} \\ x & \text{otherwise} \end{cases}$$

Note that $\overset{\geq}{\rightarrow}_{e(\mathcal{R})} = \rightarrow_{e(\mathcal{R})}$ for left-linear TRSs \mathcal{R} .

Definition 7. The TRS \mathcal{R} is called *e-raise-bounded* for L if there exists a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms in $\overset{\geq}{\rightarrow}_{e(\mathcal{R})}^*(\text{lift}_0(L))$ is at most c .

For left-linear TRSs, *e-raise-boundedness* coincides with *e-boundedness*.

Lemma 8. Let \mathcal{R} be a TRS over a signature \mathcal{F} . If $s \rightarrow_{\mathcal{R}} t$ then for all terms s' with $\text{base}(s') = s$ there exists a term t' such that $\text{base}(t') = t$ and $s' \overset{\geq}{\rightarrow}_{e(\mathcal{R})} t'$.

Proof. Straightforward. □

Theorem 9. Let \mathcal{R} be a TRS over a signature \mathcal{F} and let $L \subseteq \mathcal{T}(\mathcal{F})$. If \mathcal{R} is *top-raise-bounded* or *roof-raise-bounded* for L then \mathcal{R} is *terminating* on L . If \mathcal{R} is *non-duplicating* and *match-raise-bounded* for L then \mathcal{R} is *terminating* on L .

Proof. Assume to the contrary that there exists an infinite sequence $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} \dots$ with $t_1 \in L$. With help of Lemma 8 this sequence is lifted to an infinite $\overset{\geq}{\rightarrow}_{e(\mathcal{R})}$ sequence starting from $\text{lift}_0(t_1)$. Since \mathcal{R} is *e-raise-bounded* for L , all terms in this latter sequence belong to $\mathcal{T}(\mathcal{F}_{\{0, \dots, c\}})$ for some $c \in \mathbb{N}$. Hence the employed rules must come from $e_c(\mathcal{R}) \cup \text{raise}_c(\mathcal{F})$ and therefore $e_c(\mathcal{R}) \cup \text{raise}_c(\mathcal{F})$ is non-terminating. This is impossible because $e(\mathcal{R}) \cup \text{raise}(\mathcal{F})$ is locally terminating according to Lemma 4. □

We conclude this section with an example.

Example 10. Consider the TRS \mathcal{R} consisting of the rules $f(x, x) \rightarrow f(a, g(a, x))$ and $g(x, x) \rightarrow b$ over the signature $\mathcal{F} = \{a, f, g\}$. With the rules

$$f_0(x, x) \rightarrow f_1(a_1, g_1(a_1, x)) \qquad g_0(x, x) \rightarrow b_1 \qquad g_1(x, x) \rightarrow b_2$$

of $\text{match}(\mathcal{R})$, arbitrary derivations in \mathcal{R} can be simulated using the relation $\overset{\geq}{\rightarrow}_{\text{match}(\mathcal{R})}$. For instance,

$$\begin{aligned}
 f(f(a, a), f(a, b)) &\rightarrow_{\mathcal{R}} f(f(a, g(a, a)), f(a, b)) \\
 &\rightarrow_{\mathcal{R}} f(f(a, b), f(a, b)) \\
 &\rightarrow_{\mathcal{R}} f(a, g(a, f(a, b)))
 \end{aligned}$$

is turned into

$$\begin{aligned}
 f_0(f_0(a_0, a_0), f_0(a_0, b_0)) &\stackrel{\cong}{\rightarrow}_{\text{match}(\mathcal{R})} f_0(f_1(a_1, g_1(a_1, a_0)), f_0(a_0, b_0)) \\
 &\stackrel{\cong}{\rightarrow}_{\text{match}(\mathcal{R})} f_0(f_1(a_1, b_2), f_0(a_0, b_0)) \\
 &\stackrel{\cong}{\rightarrow}_{\text{match}(\mathcal{R})} f_1(a_1, g_1(a_1, f_1(a_1, b_2)))
 \end{aligned}$$

Here the following raise rules are used implicitly to enable the application of the non-left-linear rules in $\text{match}(\mathcal{R})$:

$$\begin{array}{ll}
 a_0 \rightarrow a_1 & b_1 \rightarrow b_2 \\
 b_0 \rightarrow b_1 & f_0(x, y) \rightarrow f_1(x, y)
 \end{array}$$

It can be shown that \mathcal{R} is match-raise-bounded by 2.

4 Quasi-deterministic Tree Automata

A common approach to handle non-linearity with automata techniques is to consider *deterministic* tree automata (cf. [2,14,15]). The weaker property defined below turns out to be more suitable for our purposes. To simplify the presentation we consider tree automata without ϵ -transitions.

Definition 11. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton. For a left-hand side $l \in \text{lhs}(\Delta)$ of a transition, we denote the set $\{q \mid l \rightarrow q \in \Delta\}$ of possible right-hand sides by $Q(l)$. We call \mathcal{A} quasi-deterministic if for every $l \in \text{lhs}(\Delta)$ there exists a state $p \in Q(l)$ such that for all transitions $f(q_1, \dots, q_n) \rightarrow q \in \Delta$ and $i \in \{1, \dots, n\}$ with $q_i \in Q(l)$, the transition $f(q_1, \dots, q_{i-1}, p, q_{i+1}, \dots, q_n) \rightarrow q$ belongs to Δ . Moreover, we require that $p \in Q_f$ whenever $Q(l)$ contains a final state.

Deterministic tree automata are trivially quasi-deterministic because $Q(l)$ is a singleton set for every left-hand side $l \in \text{lhs}(\Delta)$. In general, $Q(l)$ may contain more than one state that satisfies the above property. In the following we assume that there is a unique designated state, which we denote by p_l . The set of all designated states is denoted by Q_d and the restriction of Δ to transition rules $l \rightarrow q$ that satisfy $q = p_l$ is denoted by Δ_d .

Lemma 12. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton. If $t \rightarrow_{\Delta}^* q$ then $t \rightarrow_{\Delta_d}^* \cdot \rightarrow_{\Delta} q$ for all terms $t \in \mathcal{T}(\mathcal{F})$ and states $q \in Q$.

Proof. We use induction on t . If t is a constant the claim holds trivially. Let $t = f(t_1, \dots, t_n)$. The sequence from t to q can be written as $t \rightarrow_{\Delta}^* f(q_1, \dots, q_n) \rightarrow_{\Delta} q$. The induction hypothesis yields for every $i \in \{1, \dots, n\}$ a left-hand side $l_i \in$

lhs(Δ) such that $t_i \xrightarrow{*}_{\Delta_d} l_i \rightarrow_{\Delta} q_i$. Since \mathcal{A} is quasi-deterministic, $l_i \rightarrow_{\Delta_d} p_{l_i}$ and $q_i \in Q(l_i)$. According to the definition of p_{l_i} the transition $f(p_{l_1}, q_2, \dots, q_n) \rightarrow q$ belongs to Δ . Repeating this argument $n - 1$ times yields that the transition $f(p_{l_1}, \dots, p_{l_n}) \rightarrow q$ belongs to Δ . Thus $t \xrightarrow{*}_{\Delta_d} f(p_{l_1}, \dots, p_{l_n}) \rightarrow_{\Delta} q$. \square

Lemma 13. *Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton. The tree automaton $\mathcal{A}_d = (\mathcal{F}, Q, Q_f, \Delta_d)$ is deterministic and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_d)$.*

Proof. From the definition it is obvious that \mathcal{A}_d is deterministic. The inclusion $\mathcal{L}(\mathcal{A}_d) \subseteq \mathcal{L}(\mathcal{A})$ is trivial. In order to show the reverse inclusion, we prove the following claim for all terms $t \in \mathcal{T}(\mathcal{F})$ and states $q \in Q$:

If $t \xrightarrow{*}_{\Delta} q$ then $t \xrightarrow{*}_{\Delta_d} p_l$ and $q \in Q(l)$ for some $l \in \text{lhs}(\Delta)$.

We use induction on t . If t is a constant then $t \rightarrow q \in \Delta$. Hence $t \in \text{lhs}(\Delta)$, $q \in Q(t)$, and $t \rightarrow p_t \in \Delta_d$. Let $t = f(t_1, \dots, t_n)$. The sequence from t to q can be written as $t \xrightarrow{*}_{\Delta} f(q_1, \dots, q_n) \rightarrow_{\Delta} q$. From the previous lemma we know that $t \xrightarrow{*}_{\Delta_d} f(p_1, \dots, p_n) \rightarrow_{\Delta} q$. Let $l = f(p_1, \dots, p_n)$. We have $l \in \text{lhs}(\Delta)$, $q \in Q(l)$, and $l \rightarrow p_l \in \Delta_d$. It follows that $t \xrightarrow{*}_{\Delta_d} p_l$. This completes the proof of the claim. Now let $t \in \mathcal{L}(\mathcal{A})$. So $t \xrightarrow{*}_{\Delta} q_f$ for some $q_f \in Q_f$. From the claim we obtain $t \xrightarrow{*}_{\Delta_d} p_l$ and $q_f \in Q(l)$ for some $l \in \text{lhs}(\Delta)$. Since $Q(l)$ contains a final state, we have $p_l \in Q_f$ by definition. Hence $t \in \mathcal{L}(\mathcal{A}_d)$. \square

A simple procedure to turn an arbitrary tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ into an equivalent quasi-deterministic one *without losing any transitions of Δ* is the following:

1. Use the subset construction to transform \mathcal{A} into a deterministic tree automaton $\mathcal{A}' = (\mathcal{F}, Q', Q'_f, \Delta')$.
2. Take the union of \mathcal{A} and \mathcal{A}' after identifying states $\{q\} \in Q'$ with $q \in Q$.

Let us illustrate this on a small example.

Example 14. The tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ with $\mathcal{F} = \{a, f\}$, $Q = \{1, 2\}$, $Q_f = \{1\}$, and $\Delta = \{a \rightarrow 1, a \rightarrow 2, f(1, 2) \rightarrow 1\}$ is not quasi-deterministic; we have $Q(a) = \{1, 2\}$ but if we take $p_a = 1$ then the transition $f(1, 1) \rightarrow 1$ is missing and if we take $p_a = 2$ then the transition $f(2, 2) \rightarrow 1$ is missing. The subset construction produces $\mathcal{A}' = (\mathcal{F}, Q', Q'_f, \Delta')$ with $Q' = \{\{1\}, \{2\}, \{1, 2\}\}$, $Q'_f = \{\{1\}, \{1, 2\}\}$, and Δ' consisting of the following transitions:

$$\begin{array}{lll} a \rightarrow \{1, 2\} & f(\{1\}, \{2\}) \rightarrow \{1\} & f(\{1, 2\}, \{2\}) \rightarrow \{1\} \\ & f(\{1\}, \{1, 2\}) \rightarrow \{1\} & f(\{1, 2\}, \{1, 2\}) \rightarrow \{1\} \end{array}$$

Combining \mathcal{A} and \mathcal{A}' after identifying $\{1\}$ with 1 and $\{2\}$ with 2 produces the following transitions:

$$\begin{array}{lll} a \rightarrow \{1, 2\} & f(1, 2) \rightarrow 1 & f(\{1, 2\}, 2) \rightarrow 1 \\ a \rightarrow 1 & f(1, \{1, 2\}) \rightarrow 1 & f(\{1, 2\}, \{1, 2\}) \rightarrow 1 \\ a \rightarrow 2 & & \end{array}$$

The final states are 1 and $\{1, 2\}$, and $p_a = \{1, 2\}$.

5 Compatibility

The reason why we prefer quasi-deterministic tree automata over deterministic automata is the importance of preserving existing transitions when constructing an automaton that satisfies the compatibility condition defined below. This will be illustrated in Example 17.

Definition 15. Let \mathcal{R} be a TRS, L a language, and $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ a quasi-deterministic tree automaton. We say that \mathcal{A} is compatible with \mathcal{R} and L if $L \subseteq \mathcal{L}(\mathcal{A})$ and for each rewrite rule $l \rightarrow r \in \mathcal{R}$ and state substitution $\sigma: \text{Var}(l) \rightarrow Q_d$ such that $l\sigma \rightarrow_{\Delta_d}^* q$ it holds that $r\sigma \rightarrow_{\Delta}^* q$.

Theorem 16. Let \mathcal{R} be a TRS and L a language. Let \mathcal{A} be a quasi-deterministic tree automaton. If \mathcal{A} is compatible with \mathcal{R} and L then $\rightarrow_{\mathcal{R}}^*(L) \subseteq \mathcal{L}(\mathcal{A})$.

Proof. Let s and t be two ground terms such that $s \in \mathcal{L}(\mathcal{A})$ and $s \rightarrow_{\mathcal{R}} t$. We show that $t \in \mathcal{L}(\mathcal{A})$. The desired result then follows by induction. There exist a rewrite rule $l \rightarrow r \in \mathcal{R}$, a position $p \in \text{Pos}(s)$, and a ground substitution σ such that $s = s[l\sigma]_p \rightarrow_{\mathcal{R}} s[r\sigma]_p = t$. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$. Because $s \in \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_d)$, there exist states $q \in Q$ and $q_f \in Q_f$ such that $s = s[l\sigma]_p \rightarrow_{\Delta_d}^* s[q]_p \rightarrow_{\Delta_d}^* q_f$. Because \mathcal{A}_d is deterministic by Lemma 13, different occurrences of $x\sigma$ in $l\sigma$ are reduced to the same state in the sequence from $s[l\sigma]_p$ to $s[q]_p$. Hence there exists a mapping $\tau: \text{Var}(l) \rightarrow Q_d$ such that $l\sigma \rightarrow_{\Delta_d}^* l\tau \rightarrow_{\Delta_d}^* q$. We have $r\sigma \rightarrow_{\Delta_d}^* r\tau \rightarrow_{\Delta_d}^* \cdot \rightarrow_{\Delta} q$ by the definition of compatibility and Lemma 12. Hence $t = s[r\sigma]_p \rightarrow_{\Delta_d}^* s[q]_p \rightarrow_{\Delta_d}^* q_f$ and thus $t \in \mathcal{L}(\mathcal{A})$. \square

Since the set $\xrightarrow{e(\mathcal{R})}^*(\text{lift}_0(L))$ need not be regular, even for left-linear \mathcal{R} and regular L [10], we cannot hope to give an exact automaton construction. The general idea [6,10] is to look for violations of the compatibility requirement: $l\sigma \rightarrow_{\Delta_d}^* q$ and not $r\sigma \rightarrow_{\Delta}^* q$ for some rewrite rule $l \rightarrow r$, state substitution $\sigma: \text{Var}(l) \rightarrow Q$, and state q . Then we add new states and transitions to the current automaton to ensure $r\sigma \rightarrow_{\Delta}^* q$. There are several ways to do this, ranging from establishing a completely new path $r\sigma \rightarrow_{\Delta}^* q$ to adding as few as possible new transitions by reusing transitions from the current automaton. After $r\sigma \rightarrow_{\Delta}^* q$ has been established, we look for further violations of compatibility. This process is repeated until a compatible automaton is obtained, which may never happen if new states are kept being added.

The following example explains why we prefer quasi-deterministic automata over deterministic ones.

Example 17. Consider the TRS $\mathcal{R} = \{f(x, x) \rightarrow f(a, b), c \rightarrow a, c \rightarrow b\}$ over the signature $\mathcal{F} = \{a, b, c, f\}$ and the initial tree automaton $\mathcal{A} = (\mathcal{F}_{\{0\}}, \{1\}, \{1\}, \Delta)$ with the following transitions:

$$a_0 \rightarrow 1 \qquad b_0 \rightarrow 1 \qquad c_0 \rightarrow 1 \qquad f_0(1, 1) \rightarrow 1$$

Suppose we look for a deterministic automaton that is compatible with $\text{match}(\mathcal{R})$ and $\text{lift}_0(\mathcal{T}(\mathcal{F}))$. Note that $\mathcal{L}(\mathcal{A}) = \text{lift}_0(\mathcal{T}(\mathcal{F}))$. Since $c_0 \rightarrow_{\text{match}(\mathcal{R})} a_1$ and

$c_0 \rightarrow 1$, we add the transition $a_1 \rightarrow 1$. Similarly, $c_0 \rightarrow_{\text{match}(\mathcal{R})} b_1$ gives rise to the transition $b_1 \rightarrow 1$. Next we consider $f_0(1, 1) \rightarrow_{\text{match}(\mathcal{R})} f_1(a_1, b_1)$ with $f_0(1, 1) \rightarrow 1$. In order to ensure $f_1(a_1, b_1) \rightarrow^* 1$ we may reuse one or both of the transitions $a_1 \rightarrow 1$ and $b_1 \rightarrow 1$. Let us consider the various alternatives.

- If we reuse both transitions then we only need to add the transition $f_1(1, 1) \rightarrow 1$ in order to obtain $f_1(a_1, b_1) \rightarrow^* 1$. This gives rise to a further violation of compatibility, $f_1(1, 1) \rightarrow_{\text{match}(\mathcal{R})} f_2(a_2, b_2)$ with $f_1(1, 1) \rightarrow 1$, which is similar to the previous one.
- Suppose we reuse $a_1 \rightarrow 1$ but not $b_1 \rightarrow 1$. That means we have to add a new state 2 and transitions $b_1 \rightarrow 2$ and $f_1(1, 2) \rightarrow 1$ resulting in the following transitions:

$$\begin{array}{cccc} a_0 \rightarrow 1 & b_0 \rightarrow 1 & c_0 \rightarrow 1 & f_0(1, 1) \rightarrow 1 \\ a_1 \rightarrow 1 & b_1 \rightarrow 1 & b_1 \rightarrow 2 & f_1(1, 2) \rightarrow 1 \end{array}$$

Making these transitions deterministic produces an automaton that includes $c_0 \rightarrow 1$ and $b_1 \rightarrow \{1, 2\}$. Because the transition $b_1 \rightarrow 1$ was removed, the second violation of compatibility that we considered, $c_0 \rightarrow_{\text{match}(\mathcal{R})} b_1$ and $c_0 \rightarrow 1$, reappears. So we have to add $b_1 \rightarrow 1$ again, but each time we make the automaton deterministic this transition is deleted.

- The remaining options would be to choose a fresh state for a_1 or for both a_1 and b_1 . However they all give rise to the same situation.

So by using deterministic automata we will never achieve compatibility. The problem is clearly the removal of transitions that were added in an earlier stage to ensure compatibility and that is precisely the reason why we introduced quasi-deterministic automata. Starting from the transitions in the last case above, the following quasi-deterministic tree automaton is constructed:

$$\begin{array}{cccc} a_0 \rightarrow 1 & b_0 \rightarrow 1 & c_0 \rightarrow 1 & f_0(1, 1) \rightarrow 1 \\ a_1 \rightarrow 1 \mid 2 \mid 4 & b_1 \rightarrow 1 \mid 3 \mid 5 & & f_1(2, 3) \rightarrow 1 \\ f_0(1, 4) \rightarrow 1 & f_0(1, 5) \rightarrow 1 & f_0(4, 5) \rightarrow 1 & f_0(4, 4) \rightarrow 1 \\ f_0(4, 1) \rightarrow 1 & f_0(5, 1) \rightarrow 1 & f_0(5, 4) \rightarrow 1 & f_0(5, 5) \rightarrow 1 \\ f_1(2, 5) \rightarrow 1 & f_1(4, 3) \rightarrow 1 & f_1(4, 5) \rightarrow 1 & \end{array}$$

Here 4 (abbreviating $\{1, 2\}$) is the designated state for a_1 and 5 (abbreviating $\{1, 3\}$) is the designated state for b_1 . The transitions in the last three rows are added to satisfy the condition of Definition [11](#). The resulting automaton is compatible with $\text{match}(\mathcal{R})$.

To conclude match-raise-boundedness in the previous example, it is not enough to construct a tree automaton that is compatible with $\text{match}(\mathcal{R})$. We also have to ensure that the automaton is closed under the implicit raise steps caused by $\xrightarrow{\geq}_{\text{match}(\mathcal{R})}$. How this can be done is explained in the next section.

6 Raise-Consistency

A naive (and sound) approach to guarantee that the implicit raise rules in the definition of $\xrightarrow{\geq}_e(\mathcal{R})$ are taken into account would be to require compatibility with all raise rules $f_i(x_1, \dots, x_n) \rightarrow f_{i+1}(x_1, \dots, x_n)$ for which f_{i+1} appears in the current set of transitions. The following example shows that this approach may over-approximate the essential raise steps too much.

Example 18. Continuing the previous example, we have $f_0(1, 1) \rightarrow_{\text{raise}(\mathcal{F})} f_1(1, 1)$ with $f_0(1, 1) \rightarrow 1$. Compatibility requires the addition of the transition $f_1(1, 1) \rightarrow 1$, causing a new compatibility violation $f_1(1, 1) \rightarrow_{\text{match}(\mathcal{R})} f_2(a_2, b_2)$ with $f_1(1, 1) \rightarrow 1$. After establishing the path $f_2(a_2, b_2) \rightarrow^* 1$, f_2 will make its appearance and thus we have to consider $f_1(1, 1) \rightarrow_{\text{raise}(\mathcal{F})} f_2(1, 1)$ with $f_1(1, 1) \rightarrow 1$. This yields the transition $f_2(1, 1) \rightarrow 1$. Clearly, this process will not terminate.

To avoid the behaviour in the previous example, we now outline a better way to handle the raise rules. Let $f_i(q_1, \dots, q_n) \rightarrow q$ be a transition that we add to the current set Δ of transitions, either to resolve a compatibility violation or to satisfy the quasi-determinism condition. Then, for every transition $f_j(q_1, \dots, q_n) \rightarrow p \in \Delta$ with $j < i$ we add $f_i(q_1, \dots, q_n) \rightarrow p$ to Δ and for every transition $f_j(q_1, \dots, q_n) \rightarrow p \in \Delta$ with $j > i$ we add $f_j(q_1, \dots, q_n) \rightarrow q$ to Δ . The automata resulting from this implicit handling of raise rules satisfy the property defined below.

Definition 19. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a tree automaton with N a finite subset of \mathbb{N} . We say that \mathcal{A} is raise-consistent if for every pair of transitions $f_i(q_1, \dots, q_n) \rightarrow q$ and $f_j(q_1, \dots, q_n) \rightarrow p$ in Δ with $i < j$, the transition $f_j(q_1, \dots, q_n) \rightarrow q$ belongs to Δ .

Lemma 20. Let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$ be a quasi-deterministic tree automaton. If \mathcal{A} is raise-consistent then for all terms $s, t \in \mathcal{T}(\mathcal{F}_N)$ and states $p, q \in Q$ with $\text{base}(s) = \text{base}(t)$, $s \rightarrow_{\Delta}^* p$, and $t \rightarrow_{\Delta}^* q$ there exists a left-hand side $l \in \text{lhs}(\Delta)$ such that $s \uparrow t \rightarrow_{\Delta_d}^* l$ and $p, q \in Q(l)$.

Proof. We prove the lemma by induction on s and t . If s and t are constants then $s \uparrow t \in \{s, t\}$. If $s \leq t$ then $s \uparrow t = t$ and $p \in Q(t)$ by the definition of raise-consistency. If $t \leq s$ then $s \uparrow t = s$ and $q \in Q(s)$. So in both cases we can take $l = s \uparrow t$. For the induction step suppose that $s = f_j(s_1, \dots, s_n)$ and $t = f_k(t_1, \dots, t_n)$ with $s \rightarrow_{\Delta}^* f_j(p_1, \dots, p_n) \rightarrow_{\Delta} p$ and $t \rightarrow_{\Delta}^* f_k(q_1, \dots, q_n) \rightarrow_{\Delta} q$. The induction hypothesis yields left-hand sides $l_1, \dots, l_n \in \text{lhs}(\Delta)$ such that $s_i \uparrow t_i \rightarrow_{\Delta_d}^* l_i$ with $p_i, q_i \in Q(l_i)$ for all $i \in \{1, \dots, n\}$. Let $m = \max\{j, k\}$. Clearly $s \uparrow t = f_m(s_1 \uparrow t_1, \dots, s_n \uparrow t_n)$. Let $l = f_m(p_{l_1}, \dots, p_{l_n})$. We have $s \uparrow t \rightarrow_{\Delta_d}^* f_m(l_1, \dots, l_n) \rightarrow_{\Delta_d}^* l$. Because \mathcal{A} is quasi-deterministic, $f_j(p_{l_1}, \dots, p_{l_n}) \rightarrow p$ and $f_k(p_{l_1}, \dots, p_{l_n}) \rightarrow q$ belong to Δ . It follows that $l \in \text{lhs}(\Delta)$. Raise-consistency yields $p, q \in Q(l)$. \square

Theorem 21. Let \mathcal{R} be a TRS and L a language. Let \mathcal{A} be a quasi-deterministic and raise-consistent tree automaton. If \mathcal{A} is compatible with $e(\mathcal{R})$ and $\text{lift}_0(L)$ then \mathcal{R} is e -raise-bounded for L .

Proof. Let \mathcal{F} be the signature of \mathcal{R} and let $\mathcal{A} = (\mathcal{F}_N, Q, Q_f, \Delta)$. We have $\text{lift}_0(L) \subseteq L(\mathcal{A})$. Let $s \in L(\mathcal{A})$ and $s \xrightarrow{\geq}_{e(\mathcal{R})} t$. Then there is a term s' such that $s \xrightarrow{*}_{\text{raise}(\mathcal{F})} s' \xrightarrow{e(\mathcal{R})} t$. We show that $s' \in L(\mathcal{A})$. If l is linear then $s = s'$ and we are done. Suppose l is non-linear. To simplify the notation we assume that $l = f(x, x)$. We may write $s = s[f(s_1, s_2)]_p$ and $s' = s[f(u, u)]_p$ with $\text{base}(s_1) = \text{base}(s_2)$ and $u = s_1 \uparrow s_2$. Since $s \in L(\mathcal{A})$, there exist states $p_1, p_2, q \in Q$ and $q_f \in Q_f$ such that $s \xrightarrow{*}_{\Delta} s[f(p_1, p_2)]_p \rightarrow_{\Delta} s[q]_p \xrightarrow{*}_{\Delta} q_f$. In order to conclude $s' \in L(\mathcal{A})$ we show that $f(u, u) \xrightarrow{*}_{\Delta} q$. The previous lemma yields a left-hand side $l \in \text{lhs}(\Delta)$ such that $u \xrightarrow{*}_{\Delta_d} l$ and $p_1, p_2 \in Q(l)$. We obtain $f(u, u) \xrightarrow{*}_{\Delta_d} f(l, l) \xrightarrow{*}_{\Delta_d} f(p_1, p_1)$. Quasi-determinism yields $f(p_1, p_1) \rightarrow q \in \Delta$ and thus $f(u, u) \xrightarrow{*}_{\Delta} q$ as desired. Now that $s' \in L(\mathcal{A})$ is established, we obtain $t \in L(\mathcal{A})$ from the compatibility of \mathcal{A} and $e(\mathcal{R})$, as in the proof of Theorem 16. \square

Example 22. Since the resulting quasi-deterministic tree automaton in Example 17 is raise-consistent and compatible with $\text{match}(\mathcal{R})$ and $\text{lift}_0(\mathcal{T}(\mathcal{F}))$, \mathcal{R} is match-raise-bounded by Theorem 21.

7 Forward Closures

When proving termination of a TRS \mathcal{R} that is non-overlapping [11] or right-linear [3] it is sufficient to restrict attention to the set $\text{RFC}(\mathcal{R})$ of right-hand sides of forward closures. This set is defined as the closure of the right-hand sides of the rules in \mathcal{R} under variable renaming and narrowing. More formally, $\text{RFC}(\mathcal{R})$ is the least extension of $\text{rhs}(\mathcal{R})$ such that

- $t[r]_p \sigma \in \text{RFC}(\mathcal{R})$ whenever $t \in \text{RFC}(\mathcal{R})$ and there exist a position $p \in \mathcal{FPos}(t)$ and a fresh variant $l \rightarrow r$ of a rewrite rule in \mathcal{R} with σ a most general unifier of $t|_p$ and l ,
- $t\sigma \in \text{RFC}(\mathcal{R})$ whenever $t \in \text{RFC}(\mathcal{R})$ and σ is a variable renaming.

Dershowitz [3] obtained the following result.

Theorem 23. *A right-linear TRS \mathcal{R} is terminating if and only if \mathcal{R} is terminating on $\text{RFC}(\mathcal{R})$.* \square

The following concept has been introduced in [10]. It enables the simulation of narrowing in the definition of right-hand sides of forward closures by rewriting. This makes it possible to use tree automata to compute an approximation of $\text{RFC}(\mathcal{R})$ for linear \mathcal{R} .

Definition 24. *Let \mathcal{R} be a TRS. The TRS $\mathcal{R}_{\#}$ is defined as the least extension of \mathcal{R} that is closed under the following operation. If $l \rightarrow r \in \mathcal{R}_{\#}$ and $p \in \mathcal{FPos}(l) \setminus \{\epsilon\}$ then $l[\#]_p \rightarrow r\sigma \in \mathcal{R}_{\#}$. Here the substitution σ is defined by $\sigma(x) = \#$ if $x \in \text{Var}(l|_p)$ and $\sigma(x) = x$ otherwise. The substitution that maps all variables to $\#$ is denoted by $\sigma_{\#}$. Here $\#$ is a fresh function symbol.*

The following results are proved in [10].

Lemma 25. *If \mathcal{R} is a linear TRS then $\text{RFC}(\mathcal{R})\sigma_{\#} = \rightarrow_{\mathcal{R}_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#})$. \square*

Corollary 26. *If a linear TRS \mathcal{R} is match-bounded for $\rightarrow_{\mathcal{R}_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#})$ then \mathcal{R} is terminating. \square*

In order to obtain a corresponding result for right-linear TRSs, we linearize left-hand sides of rewrite rules.

Definition 27. *Let t be a term. The set of linear terms s with $\text{Var}(t) \subseteq \text{Var}(s)$ for which there exists a variable substitution $\tau: \text{Var}(s) \setminus \text{Var}(t) \rightarrow \text{Var}(t)$ such that $s\tau = t$ is denoted by $\text{linear}(s)$. Let \mathcal{R} be a TRS. The set of rewrite rules $\{l' \rightarrow r \mid l \rightarrow r \in \mathcal{R} \text{ and } l' \in \text{linear}(l)\}$ is denoted by $\text{linear}(\mathcal{R})$. We write $\mathcal{R}'_{\#}$ for $\text{linear}(\mathcal{R})_{\#}$.*

Lemma 28. *If \mathcal{R} is right-linear then $\text{RFC}(\mathcal{R})\sigma_{\#} \subseteq \rightarrow_{\mathcal{R}'_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#})$.*

Proof. We obviously have $\text{rhs}(\mathcal{R}) = \text{rhs}(\text{linear}(\mathcal{R}))$. Applying Lemma 25 to $\text{linear}(\mathcal{R})$ yields $\text{RFC}(\text{linear}(\mathcal{R}))\sigma_{\#} = \rightarrow_{\mathcal{R}'_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#})$. Hence it is sufficient to prove the inclusion $\text{RFC}(\mathcal{R})\sigma_{\#} \subseteq \text{RFC}(\text{linear}(\mathcal{R}))\sigma_{\#}$. We omit the straightforward details. \square

The following example shows that the reverse inclusion does not hold.

Example 29. For the TRS $\mathcal{R} = \{f(x, x) \rightarrow f(b, g(x)), a \rightarrow b\}$ we have $\text{RFC}(\mathcal{R})\sigma_{\#} = \{f(b, g(\#)), b\}$ and $\rightarrow_{\mathcal{R}'_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#}) = \{f(b, g^i(\#)), b, f(b, g^i(b)) \mid i \geq 1\}$.

Corollary 30. *Let \mathcal{R} be a right-linear TRS. If \mathcal{R} is match-raise-bounded for $\rightarrow_{\mathcal{R}'_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#})$ then \mathcal{R} is terminating.*

Proof. Since $\text{RFC}(\mathcal{R})\sigma_{\#}$ is a subset of $\rightarrow_{\mathcal{R}'_{\#}}^*(\text{rhs}(\mathcal{R})\sigma_{\#})$, \mathcal{R} is also match-raise-bounded for $\text{RFC}(\mathcal{R})\sigma_{\#}$. Theorem 9 yields the termination of \mathcal{R} on $\text{RFC}(\mathcal{R})\sigma_{\#}$. Since rewriting is closed under substitution, \mathcal{R} is terminating on $\text{RFC}(\mathcal{R})$. From Theorem 23 we conclude that \mathcal{R} is terminating on all terms. \square

We conclude this section by stating a simple criterion that allows us to restrict the set of terms that have to be considered for termination of TRSs that are non-duplicating but not necessarily right-linear. The easy proof is omitted. For right-linear systems the use of forward closures is more powerful.

Lemma 31. *Let \mathcal{R} be a non-duplicating TRS over a signature \mathcal{F} and let $\mathcal{G} \subseteq \mathcal{F}$ consist of all function symbols in $\text{rhs}(\mathcal{R})$. Then \mathcal{R} is terminating if and only if \mathcal{R} is terminating on $\mathcal{T}(\mathcal{G})$. \square*

8 Experimental Results

The techniques described in the preceding sections are implemented in $\mathsf{T}\overline{\mathsf{T}}\mathsf{box}$.¹ $\mathsf{T}\overline{\mathsf{T}}\mathsf{box}$ is written in OCaml² and consists of about 5000 lines of code.

¹ <http://cl-informatik.uibk.ac.at/~mkorp/TTTbox.html>

² <http://caml.inria.fr/>

Table 1. 87 non-left-linear TRSs

	28 right-linear				59 non-right-linear	
			RFC		explicit	implicit
	explicit	implicit	explicit	implicit		
# successes	8	9	21	21	4	8
average time	3	4	6	6	3421	549
# timeouts	20	19	7	7	55	51

Since quasi-determinisation is expensive, $\mathsf{T}\mathsf{T}\mathsf{box}$ collects and resolves all compatibility violations with respect to the current automaton before making the automaton quasi-deterministic. Then new compatibility violations are determined and the process is repeated. Compatibility violations are resolved by adding new transitions according to the following strategy, which is a variation of the one used by Matchbox [16]. To establish a path $r\sigma \rightarrow_{\Delta}^* q$, $\mathsf{T}\mathsf{T}\mathsf{box}$

1. calculates all contexts $C[\square, \dots, \square]$, $D_1[\square, \dots, \square], \dots, D_n[\square, \dots, \square]$ and terms $t_1, \dots, t_m \in \mathcal{T}(\mathcal{F}, \mathcal{Q})$ such that $C[D_1[t_1, \dots, t_i], \dots, D_n[t_j, \dots, t_m]] = r\sigma$, $C[q_1, \dots, q_n] \rightarrow_{\Delta}^* q$ and $t_i \rightarrow_{\Delta}^* q_{t_i}$ for states $q_1, \dots, q_n, q_{t_1}, \dots, q_{t_m} \in \mathcal{Q}$,
2. chooses among all possibilities one where the combined size of $D_1[\square, \dots, \square], \dots, D_n[\square, \dots, \square]$ is minimal,
3. adds new transitions involving new states to achieve $D_1[q_{t_1}, \dots, q_{t_i}] \rightarrow^* q_1, \dots, D_n[q_{t_j}, \dots, q_{t_m}] \rightarrow^* q_n$.

We report on the experiments we performed with $\mathsf{T}\mathsf{T}\mathsf{box}$ on the 87 non-left-linear TRSs in version 3.2 of the Termination Problem Data Base.³ All tests were performed on a server equipped with two Intel® Xeon™ processors running at a CPU rate of 2.40 GHz and 2048 MB of system memory. Our results are summarized in Table 1. We list the number of successful termination attempts, the average system time needed to prove termination (measured in milliseconds), and the number of timeouts. For all experiments we used a 60 seconds time limit. In the left part of the table we deal with right-linear systems and test for match-raise-boundedness, both with the explicit approach for handling raise rules described in the first paragraph of Section 6 and the implicit approach using raise-consistency. The positive effect of forward closures (Corollary 30) is clearly visible. In the right part of Table 1 we deal with non-right-linear TRSs. If the TRS under consideration is non-duplicating we test for match-raise-boundedness; duplicating TRSs are tested for roof-raise-boundedness.

All non-left-linear TRSs in the Termination Problem Data Base that can be proved terminating with $\mathsf{T}\mathsf{T}\mathsf{box}$ can also be proved terminating with other termination provers. Nevertheless there are non-left-linear TRSs which can currently only be handled by $\mathsf{T}\mathsf{T}\mathsf{box}$. One such TRS consists of the following rules:

³ <http://www.lri.fr/~marche/tpdb>

$$\begin{array}{ll}
 f(g(x, y)) \rightarrow g(y, g(f(f(x)), a)) & g(c, g(c, x)) \rightarrow g(e, g(d, x)) \\
 g(x, x) \rightarrow g(a, b) & g(d, g(d, x)) \rightarrow g(c, g(e, x)) \\
 & g(e, g(e, x)) \rightarrow g(d, g(c, x))
 \end{array}$$

Using the implicit approach, $\Upsilon\Upsilon\text{box}$ certifies in 0.059 seconds that this TRS is match-raise-bounded for the set of right-hand sides of forward closures by 2. It produces a quasi-deterministic and raise-consistent tree automaton that has 47 states and 213 transitions. None of the tools that participated in last year's termination competition can prove termination within 300 seconds.

9 Two Results for Match-Bounded String Rewriting

For the class of string rewrite systems (SRSs in the following), the match-bound technique has some properties which do not hold for the more general case of (left-linear) term rewriting. One of these is regularity preservingness [7]. The following result of [7] states that match-boundedness is decidable if the bound c is fixed.

Theorem 32. *The following problem is decidable:*

instance: an SRS \mathcal{R} , a regular set L , a bound $c \in \mathbb{N}$
question: is \mathcal{R} match-bounded for L by c ?

An efficient and exact algorithm for finding a compatible automaton that solves the problem of Theorem 32 is described in Endrullis *et al.* [5] and implemented in **Jambox**. When c is not fixed, match-boundedness becomes undecidable. This settles an open problem in [7].⁴

Theorem 33. *The following problem is undecidable:*

instance: an SRS \mathcal{R} and a regular set L
question: is \mathcal{R} match-bounded for L ?

Proof. Let t be an arbitrary string and consider $L = \{t\}$. We show that \mathcal{R} is match-bounded for L if and only if t is terminating. Since the termination problem for SRSs is undecidable [13], the result follows. If \mathcal{R} is match-bounded for L then t is terminating according to [7, Theorem 2]. Otherwise, for each $c \in \mathbb{N}$ there exists a string u and a derivation $\text{lift}_0(t) \rightarrow_{\text{match}(\mathcal{R})}^* u$ such that u contains a function symbol of height c . Because the relation $\rightarrow_{\text{match}(\mathcal{R})}$ is finitely branching, it follows that there must be an infinite $\text{match}(\mathcal{R})$ -derivation starting from $\text{lift}_0(t)$. Erasing the heights from this derivation produces an infinite \mathcal{R} -derivation from t . Hence t is not terminating. \square

The properties e -boundedness and e -raise-boundedness for $e \in \{\text{match}, \text{roof}, \text{top}\}$ are likewise undecidable. In order to prove that a given SRS \mathcal{R} is *not* match-bounded for a given set L , the following concept was introduced in [7].

⁴ The question whether an SRS is match-bounded for the set of all strings remains open.

Definition 34. Let \mathcal{R} be an SRS and W a nonempty set that does not contain the empty string. The set of all strings t for which there exist a string $s \in W$ and strings u, t', v such that $\text{lift}_0(s) \xrightarrow{*}_{\text{match}(\mathcal{R})} ut'v$, $\text{base}(t') = t$, and the height of every symbol in t' is at least 1, is denoted by $\text{raised}(\mathcal{R}, W)$. If $W \subseteq \text{raised}(\mathcal{R}, W)$ then W is called a witnessing set for \mathcal{R} .

Lemma 35 ([7]). Let \mathcal{R} be an SRS such that $\epsilon \notin \text{lhs}(\mathcal{R})$. If W is a witnessing set for \mathcal{R} then \mathcal{R} is not match-bounded for W . □

In [7] it is further shown that an SRS is *looping* if and only if it admits a *finite* witnessing set. We now show that the converse of Lemma 35 does not hold. We do this by exhibiting an SRS \mathcal{R} without witnessing set that is not match-bounded (for the set of all strings). This settles an open problem in [7].

Lemma 36. The SRS $\mathcal{R} = \{aab \rightarrow ab, bc \rightarrow ab\}$ is not match-bounded for $\{a, b, c\}^*$.

Proof. Similar to the proof of Claim 2 in [7, Example 20]. First we prove by induction on n that

$$a_1 b_1 c_0^{2^n - 1} \xrightarrow{*}_{\text{match}(\mathcal{R})} a_{n+1} b_{n+1}$$

for all $n \geq 1$. If $n = 1$ then $a_1 b_1 c_0 \xrightarrow{\text{match}(\mathcal{R})} a_1 a_1 b_1 \xrightarrow{\text{match}(\mathcal{R})} a_2 b_2$. If $n > 1$ then

$$\begin{aligned} a_1 b_1 c_0^{2^n - 1} &= a_1 b_1 c_0^{2^{n-1} - 1} c_0^{2^{n-1}} \xrightarrow{*}_{\text{match}(\mathcal{R})} a_n b_n c_0^{2^n - 1} \xrightarrow{\text{match}(\mathcal{R})} a_n a_1 b_1 c_0^{2^{n-1} - 1} \\ &\xrightarrow{*}_{\text{match}(\mathcal{R})} a_n a_n b_n \xrightarrow{*}_{\text{match}(\mathcal{R})} a_{n+1} b_{n+1} \end{aligned}$$

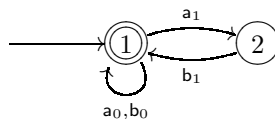
by applying the induction hypothesis twice. It follows that

$$a_0 b_0 c_0^{2^n} \xrightarrow{\text{match}(\mathcal{R})} a_0 a_1 b_1 c_0^{2^n - 1} \xrightarrow{\text{match}(\mathcal{R})} a_1 b_1 c_0^{2^{n-1} - 1} \xrightarrow{*}_{\text{match}(\mathcal{R})} a_{n+1} b_{n+1}$$

for all $n \geq 1$ and hence \mathcal{R} is not match-bounded. □

Lemma 37. The SRS $\mathcal{R} = \{aab \rightarrow ab, bc \rightarrow ab\}$ admits no witnessing set.

Proof. Assume to the contrary that $W \subseteq \text{raised}(\mathcal{R}, W)$ for some nonempty set $W \subseteq \{a, b, c\}^+$. Since c does not appear in any right-hand side of \mathcal{R} it can never reach a height greater than 0. Hence no string in W contains c and thus the rule $bc \rightarrow ab$ cannot be used in establishing the inclusion $W \subseteq \text{raised}(\mathcal{R}, W)$. It follows that $W \subseteq \text{raised}(\mathcal{R}', W)$ for the SRS $\mathcal{R}' = \{aab \rightarrow ab\}$. The following finite automaton certifies that \mathcal{R}' is match-bounded for $\{a, b\}^*$ by 1:



Since $W \subseteq \{a, b\}^*$, \mathcal{R}' is also match-bounded for W . But then $W \subseteq \text{raised}(\mathcal{R}', W)$ cannot hold by Lemma 35. □

Acknowledgments

We thank Alfons Geser for simplifying our earlier example in Lemmata [36](#) and [37](#).

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2002) Available from www.grappa.univ-lille3.fr/tata
3. Dershowitz, N.: Termination of linear rewriting systems (preliminary version). In: Even, S., Kariv, O. (eds.) Automata, Languages and Programming. LNCS, vol. 115, pp. 448–458. Springer, Heidelberg (1981)
4. Endrullis, J.: Jambox: Automated termination proofs for string/term rewriting (2006) Available from <http://joerg.endrullis.de/>
5. Endrullis, J., Hofbauer, D., Waldmann, J.: Decomposing terminating rewrite relations. In: Proc. 8th WST, pp. 39–43 (2006)
6. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) Proc. 9th RTA. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)
7. Geser, A., Hofbauer, D., Waldmann, J.: Match-bounded string rewriting systems. AAECC 15(3-4), 149–171 (2004)
8. Geser, A., Hofbauer, D., Waldmann, J.: Termination proofs for string rewriting systems via inverse match-bounds. JAR 34(4), 365–385 (2005)
9. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: Finding finite automata that certify termination of string rewriting systems. IJFCS 16(3), 471–486 (2005)
10. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. I&C 205(4), 512–534 (2007)
11. Geupel, O.: Overlap closures and termination of term rewriting systems. Report MIP-8922, Universität Passau (1989)
12. 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)
13. Huet, G., Lankford, D.S.: On the uniform halting problem for term rewriting systems. Rapport Laboria 283, INRIA (1978)
14. Middeldorp, A.: Approximating dependency graphs using tree automata techniques. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 593–610. Springer, Heidelberg (2001)
15. Nagaya, T., Toyama, Y.: Decidability for left-linear growing term rewriting systems. I&C 178(2), 499–514 (2002)
16. Waldmann, J.: Matchbox: A tool for match-bounded string rewriting. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 85–94. Springer, Heidelberg (2004)
17. Zantema, H.: Termination of rewriting proved automatically. JAR 34, 105–139 (2005)

Sequence Unification Through Currying^{*}

Temur Kutsia¹, Jordi Levy², and Mateu Villaret³

¹ Research Institute for Symbolic Computation (RISC),

Johannes Kepler University of Linz, Linz, Austria

<http://www.risc.uni-linz.ac.at/people/tkutsia/>

² Artificial Intelligence Research Institute (IIIA),

Spanish Council for Scientific Research (CSIC), Barcelona, Spain

<http://www.iiia.csic.es/~levy>

³ Departament d'Informàtica i Matemàtica Aplicada (IMA),

Universitat de Girona (UdG), Girona, Spain

<http://ima.udg.es/~villaret>

Abstract. Sequence variables play an interesting role in unification and matching when dealing with terms in an unranked signature. Sequence Unification generalizes Word Unification and seems to be appealing for information extraction in XML documents, program transformation, and rule-based programming.

In this work we study a relation between Sequence Unification and another generalization of Word Unification: Context Unification. We introduce a variant of Context Unification, called Left-Hole Context Unification that serves us to reduce Sequence Unification to it: We define a partial currying procedure to translate sequence unification problems into left-hole context unification problems, and prove soundness of the translation. Furthermore, a precise characterization of the shape of the unifiers allows us to easily reduce Left-Hole Context Unification to (the decidable problem of) Word Unification with Regular Constraints, obtaining then a decidability proof for an extension of Sequence Unification.

1 Introduction

In this work we study a relation between Sequence and Context Unification. Both problems are generalizations of Word Unification [7,13,22,26,30]. Word Unification is the problem of solving equations between terms build up from letters and word variables. A solution of a word equation is a mapping from variables to words that when applied to both sides of the equation the result is the same word.

Sequence Unification is the problem of solving equations between terms built up using an unranked signature (aka flexible arity, or variadic function symbols)

^{*} This research has been partially funded by the CICYT research projects iDEAS (TIN2004-04343) and Mulog (TIN2004-07933-C03-01), by the Austrian Science Foundation (FWF) under Project SFB F1302, and by the EC Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIENCE (026133).

and sequence and individual variables. Sequence variables are instantiated with finite sequences of terms, while individual variables instantiate to a single term. Sequence Unification is decidable and infinitary [15,16].

Solving equations with sequence variables has quite a broad range of applications. The rule-based programming language of Mathematica [31] relies on a pattern matching mechanism, which supports sequence variables and flexible arity function symbols. It can do matching modulo certain equational theories as well. Solving equations with sequence variables form a basis for schema transformation operations [3,27] used in synthesis and transformation of logic programs. Other applications include knowledge engineering and artificial intelligence [8,11,12], automated reasoning [2,9,25], rewriting [10], functional logic programming [1]. The ISO standard proposal for Common Logic [6] has notation for sequence variables (called there sequence markers). Recently there have been developments in XML querying and transformation that model XML documents with terms over an unranked signature and use sequence matching and unification techniques [15] for querying, transforming, and verifying them [4,5]. Obviously, we can not give an exhaustive overview of all the applications here.

Context Unification is the problem of solving equations between terms built up using ranked signatures and with first-order and context variables. The latter occur as monadic function symbols and denote contexts, i.e. terms with exactly one hole. When the ranked signature considered is restricted to not contain symbols of arity greater than one, the problem is equivalent to Word Unification. When allowing one single binary symbol, its decidability is still unknown [21]. Nevertheless several fragments and variants are known to be decidable [18,19,28]. The main application field of Context Unification is computational linguistics, mainly in compositional semantics of natural language [14,19,24].

Combining sequence and context variables in a single framework and equipping it with regular constraint solving methods makes the framework more flexible, with many potential applications [17,23].

The goal of this paper is to look in depth into relations between Sequence and Context Unification. Throughout curryfication we define a translation from sequence unification problems into context unification problems over a signature consisting of constants and a single binary function symbol @ (curried context unification problems), in addition, sequence variables are “encoded” into context variables while individual variables become first-order variables. The translation preserves solvability in one direction: If the sequence unification problem is solvable, then the corresponding context unification problem is solvable. To preserve solvability in the other direction, we have to restrict possible solutions of the curried context unification problems, which leads to a new variant of Context Unification that we call Left-Hole Context Unification. We prove that Left-Hole Context Unification is a decidable variant of Context Unification. We do it by reducing Left-Hole Context Unification to Word Unification with Regular Constraints that is known to be decidable [30]. The reduction transforms context equations into word equations on the postorder traversal of the terms.

Regular constraints are required to filter the solutions of the word equations that correspond to traversals of terms. This reduction is based in some ideas of [20].

With these reductions we get a new decidability proof for Sequence Unification, easier than the one in [16]. In addition we also get decidability for an extension of Sequence Unification. Moreover, this translation also allows us to use the complexity results for context matching of [29] to characterize complexity of sequence matching.

The paper proceeds as follows: Section 2 defines the two main problems: Sequence and Context Unification, Section 3 introduces the currying encoding and shows its soundness, in Section 4 decidability of Left-Hole Context Unification is shown, Section 5 discusses some extensions of Sequence Unification thanks to the currying process, Section 6 is the conclusion.

2 Preliminary Definitions

2.1 Unranked Signatures

Given an unranked signature Σ_U (i.e., a finite set of function symbols that have flexible arity), a countable set of individual variables \mathcal{V}_I , and a countable set of sequence variables \mathcal{V}_S , we define *unranked terms over Σ_U and $\mathcal{V} = \mathcal{V}_I \cup \mathcal{V}_S$* by the following grammar:

$$r ::= v \mid V \mid f(r_1, \dots, r_n)$$

where $v \in \mathcal{V}_I$, $V \in \mathcal{V}_S$, $f \in \Sigma_U$, and $n \geq 0$. The sets Σ_U , \mathcal{V}_I and \mathcal{V}_S are mutually disjoint. We will abbreviate terms of the form $f()$ by f . The set of unranked terms over Σ_U and \mathcal{V} is denoted by $\mathcal{T}(\Sigma_U, \mathcal{V})$, or simply by \mathcal{T}_U when the signature and the set of variables are unimportant. The letters f, g, a and b will be used for function symbols, v and u for individual variables, V and U for sequence variables, w for individual or sequence variables, and r and l for unranked terms. We call unranked terms from $\mathcal{T}(\Sigma_U, \mathcal{V}) \setminus \mathcal{V}_S$ the *individual terms*.

A *substitution for individual and sequence variables (IS-substitution for short)*, is a mapping from individual variables to individual terms, and from sequence variables to finite sequences of unranked terms such that all but finitely many individual variables are mapped to themselves, and all but finitely many sequence variables are mapped to themselves considered as singleton sequences¹. We use the Greek letters σ and ϑ to denote IS-substitutions.

The *composition* of two IS-substitutions σ and ϑ , written as $\sigma \circ \vartheta$, is defined by $(\sigma \circ \vartheta)(r) = \sigma(\vartheta(r))$. Given an IS-substitution σ , we represent it as $[v_1 \mapsto \sigma(v_1), \dots, v_n \mapsto \sigma(v_n), V_1 \mapsto \sigma(V_1), \dots, V_m \mapsto \sigma(V_m)]$ where v 's and V 's are all those variables for which $\sigma(v) \neq v$ and $\sigma(V) \neq V$.

The application of an IS-substitution σ to an unranked term r , denoted $\sigma(r)$, is defined by

$$\sigma(r) := \begin{cases} \sigma(w) & \text{if } r = w \\ f(\sigma(r_1), \dots, \sigma(r_n)) & \text{if } r = f(r_1, \dots, r_n) \end{cases}$$

¹ We do not distinguish between a singleton sequence and its sole member.

Similarly, σ can be applied to a sequence of unranked terms $\langle r_1, \dots, r_n \rangle$: $\sigma(\langle r_1, \dots, r_n \rangle) = \langle \sigma(r_1), \dots, \sigma(r_n) \rangle$. For the sake of readability, we put sequences between angular brackets, that are later flattened: if $\sigma = [u \mapsto a, V \mapsto \langle f(b), c \rangle]$ then $\sigma(f(u, V)) = f(a, \langle f(b), c \rangle) = f(a, f(b), c)$. We call $\sigma(r)$ (resp. $\sigma(\langle r_1, \dots, r_n \rangle)$) an *instance* of r (resp. of $\langle r_1, \dots, r_n \rangle$) under σ . Note that the set of unranked terms is not closed under IS-substitution application: An instance of a sequence variable is an unranked term sequence that in general is not an unranked term. However, an instance of an individual term is always an individual term. An IS-substitution σ_1 is *more general* than σ_2 with respect to a set of variables W , written $\sigma_1 \preceq^W \sigma_2$, if there exists ϑ such that $(\vartheta \circ \sigma_1)(w) = \sigma_2(w)$, for each $w \in W$.

A *sequence unification problem* (SU problem) is a set of *equations* (unoriented pairs) of individual terms, denoted $\{l_1 \stackrel{?}{=} r_1, \dots, l_n \stackrel{?}{=} r_n\}$. IS-Substitutions extend to equations and unification problems: $\sigma(l_1 \stackrel{?}{=} r_1) = \sigma(l_1) \stackrel{?}{=} \sigma(r_1)$ and $\sigma(\{e_1, \dots, e_n\}) = \{\sigma(e_1), \dots, \sigma(e_n)\}$. A *unifier* of a sequence unification problem Γ is an IS-substitution σ such that $\sigma(l) = \sigma(r)$ for each $l \stackrel{?}{=} r \in \Gamma$, and Γ is *solvable* if it has a unifier. A unifier σ of Γ is called *ground*, if $\sigma(\Gamma)$ contains no variables. A unifier σ_1 of Γ is *more general* than another σ_2 , if $\sigma_1 \preceq^{vars(\Gamma)} \sigma_2$. A unifier σ is *most general*, if any other unifier σ' satisfying $\sigma' \preceq^{vars(\Gamma)} \sigma$ also satisfies $\sigma \preceq^{vars(\Gamma)} \sigma'$.

2.2 Ranked Signatures

A ranked signature Σ_R is a finite set of fixed arity function symbols. We assume that Σ_R contains the 0-ary symbol \bullet , called the *hole*. Given Σ_R , a countable set of first-order variables \mathcal{X}_F , and a countable set of context variables \mathcal{X}_C , we define *ranked terms over Σ_R and \mathcal{X}* $\mathcal{X} = \mathcal{X}_F \cup \mathcal{X}_C$ by the following grammar:

$$t ::= x \mid X(t) \mid f(t_1, \dots, t_n)$$

where $x \in \mathcal{X}_F$, $X \in \mathcal{X}_C$, $f \in \Sigma_R$ such that f is n -ary and with $n \geq 0$. When $n = 0$, we omit the parentheses and write just f . The sets Σ_R , \mathcal{X}_F and \mathcal{X}_C are mutually disjoint. *Constants* are 0-ary function symbols. The set of ranked terms over Σ_R and \mathcal{X} is denoted by $\mathcal{T}(\Sigma_R, \mathcal{X})$ or simply by \mathcal{T}_R when the signature and the set of variables are unimportant. A *context* is a ranked term with exactly one occurrence of the hole. A context C may be *applied* to a ranked term t , written $C[t]$, and the result is a ranked term over Σ_R and \mathcal{X} obtained from C by replacing the hole with t . The letters x and y will be used for first-order variables, X and Y for context variables, z for first-order or context variables, a and b for constants, f for function symbols and s and t for ranked terms. The size of a term t , noted by $|t|$, is defined as its number of symbols.

A *substitution for first-order and context variables*, or an *FC-substitution* in short, is a mapping from first-order variables to hole-free ranked terms, and from context variables to contexts such that all but finitely many first-order variables are mapped to themselves, and all but finitely many context variables are mapped to themselves applied to the hole. We use the Greek letters φ and ρ to denote them. Composition is defined as above.

Given an FC-substitution φ , we represent it as $[x_1 \mapsto \varphi(x_1), \dots, x_n \mapsto \varphi(x_n), X_1 \mapsto \varphi(X_1), \dots, X_m \mapsto \varphi(X_m)]$, where x 's are all first-order variables such that $\varphi(x) \neq x$, and X 's are all context variables such that $\varphi(X) \neq X(\bullet)$.

The application of an FC-substitution φ to a ranked term t , denoted $\varphi(t)$, is defined by

$$\varphi(t) := \begin{cases} \varphi(x) & \text{if } t = x \\ \varphi(X)[\varphi(s)] & \text{if } t = X(s) \\ f(\varphi(s_1), \dots, \varphi(s_n)) & \text{if } t = f(s_1, \dots, s_n) \end{cases}$$

A *context unification problem* (CU problem) is a set of *equations* (unoriented pairs) of ranked *hole-free* terms, denoted $\{s_1 \stackrel{?}{\approx} t_1, \dots, s_n \stackrel{?}{\approx} t_n\}$. A *unifier* of a context unification problem Δ is an FC-substitution φ such that $\varphi(s) = \varphi(t)$, for each $s \stackrel{?}{\approx} t \in \Delta$, and Δ is solvable, if it has a unifier.

The notions of a more general and most general FC-unifier are defined in the same way as for IS-unifiers.

3 Currying Terms

In this section we define the *curryfication* transformation that will serve us to transform sequence unification problems into (a variant of) context unification problems.

We firstly thought that this reduction was solvability preserving: a sequence unification problem is solvable, if, and only if, its transformation into a context unification problem is solvable. However, although the implication to the right is almost trivial (Lemma 2), while trying to prove the other direction shows us that there are solutions of the context unification problems that, when interpreted as solutions for sequence unification problems, are not valid. We find out what is the kind of solutions that we need to consider in order to get the left implication. This characterization leads us to make the reduction, not directly to Context Unification but to a variant of it, called *Left-Hole Context Unification*.

We assume that for each $f \in \Sigma_U$ there exists a unique and distinct constant $a_f \in \Sigma_R \setminus \{\bullet\}$. The set of these constants is denoted by Σ_0^C . We also assume that Σ_R contains a binary function symbol $@$ and define $\Sigma_2^C = \{@\}$. Then, the set $\Sigma_U^C = \Sigma_0^C \cup \Sigma_2^C$ is called the *curried signature* corresponding to Σ_U .

Similarly, we associate to each $v \in \mathcal{V}_1$ a unique and distinct first-order variable $x_v \in \mathcal{X}_F$, and to each $V \in \mathcal{V}_S$ a context variable $X_V \in \mathcal{X}_C$. The set of such first-order and context variables is denoted by \mathcal{V}^C and is called the *curried set of variables* corresponding to \mathcal{V} .

Definition 1. *The currying function $\mathcal{C} : \mathcal{T}(\Sigma_U, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma_U^C, \mathcal{V}^C)$ is defined recursively as follows:*

$$\begin{aligned} \mathcal{C}(f) &= a_f \\ \mathcal{C}(v) &= x_v \\ \mathcal{C}(f(r_1, \dots, r_n, V)) &= X_V(\mathcal{C}(f(r_1, \dots, r_n))) \\ \mathcal{C}(f(r_1, \dots, r_n)) &= @(\mathcal{C}(f(r_1, \dots, r_{n-1})), \mathcal{C}(r_n)), \text{ where } n > 0, \text{ and } r_n \notin \mathcal{V}_S. \end{aligned}$$

We also define $\mathcal{C}(s \stackrel{?}{=} t)$ as $\mathcal{C}(s) \stackrel{?}{\approx} \mathcal{C}(t)$ and extend it to unification problems.

Using this definition we get $\mathcal{C}(f(b, V, f(v))) = @ (X_V(@ (a_f, a_b)), @ (a_f, x_v))$.

The currying function can be extended to transform sequences of unranked terms into contexts. To do so, we extend Σ_U by a flexible arity function symbol \diamond , Σ_U^C by the hole \bullet , define $\mathcal{C}(\diamond) = \bullet$, and then define the currying function for a sequence of unranked terms (that do not contain \diamond) as $\mathcal{C}(\langle r_1, \dots, r_n \rangle) = \mathcal{C}(\diamond(r_1, \dots, r_n))$. For instance, we get $\mathcal{C}(\langle V, a, f(a, b) \rangle) = \mathcal{C}(\diamond(V, a, f(a, b))) = @ (@ (X_V(\bullet), a_a), @ (@ (a_f, a_a), a_b))$.

We can use this extension to curryify IS-substitutions to FC-substitutions: For an IS-substitution σ the corresponding $\mathcal{C}(\sigma)$ is defined as the substitution that maps each variable $\mathcal{C}(w)$ to $\mathcal{C}(\sigma(w))$ and any other variable to itself. For instance, currying $\sigma = [v \mapsto f(a, V), u \mapsto g(b), V \mapsto \langle U, a, b \rangle, U \mapsto V]$ gives $\mathcal{C}(\sigma) = [x_v \mapsto X_V(@ (a_f, a_a)), x_u \mapsto @ (a_g, a_b), X_V \mapsto @ (@ (X_U(\bullet), a_a), a_b), X_U \mapsto X_V(\bullet)]$ (assuming $X_U \neq X_V$).

Remark 1. It is interesting to notice that currying sequences of unranked terms produces contexts. Moreover the “shape” of these contexts will play a crucial role to prove the final result. In fact, the instantiations of context variables that we will consider must correspond to “curry forms” of sequences. This fact will allow us to prove that minimal solutions of the context equations resulting from currying process are rab and Strahler-bounded (see Lemmas 6 and 7 in next section), and prove that context unification restricted to this kind of unifiers is decidable.

Definition 2. Given a term $t \in \mathcal{T}(\Sigma_U^C, \mathcal{V}^C)$, we say that it is well-typed (w.r.t. Σ_U), if $\mathcal{C}^{-1}(t)$ is defined, i.e. if there exists an $r \in \mathcal{T}(\Sigma_U, \mathcal{V})$ such that $\mathcal{C}(r) = t$.

Given a context $C \in \mathcal{T}(\Sigma_U^C \cup \{\bullet\}, \mathcal{V}^C)$, we say that it is well-typed (w.r.t. Σ_U), if $\mathcal{C}^{-1}(C)$ is defined, i.e. if there exists a sequence $\langle r_1, \dots, r_n \rangle$, $r_i \in \mathcal{T}(\Sigma_U, \mathcal{V})$, $1 \leq i \leq n$, such that $\mathcal{C}(\langle r_1, \dots, r_n \rangle) = C$.

Let φ be an FC-substitution such that $\varphi(z) \in \mathcal{T}(\Sigma_U^C \cup \{\bullet\}, \mathcal{V}^C)$ for all $z \in \mathcal{V}^C$. We say that φ is well-typed (w.r.t. Σ_U), if $\varphi(z)$ is well-typed for all $z \in \mathcal{V}^C$.

Lemma 1. For any IS-substitution σ and for any unranked term (or sequence of unranked terms) r over Σ_U and \mathcal{V} , we have $\mathcal{C}(\sigma)(\mathcal{C}(r)) = \mathcal{C}(\sigma(r))$.

Proof: By structural induction on r . ■

Lemma 2. If the sequence unification problem Γ over Σ_U and \mathcal{V} is solvable, then the context unification problem $\mathcal{C}(\Gamma)$ over $\Sigma_U^C \cup \{\bullet\}$ and \mathcal{V}^C is also solvable.

Proof: Let σ be a unifier of Γ . Then, by Lemma 1, it is easy to prove that $\mathcal{C}(\sigma)$ is a unifier of $\mathcal{C}(\Gamma)$. ■

In fact with the previous lemmas we have proved a stronger result: given a unifier σ of $l \stackrel{?}{=} r$, we can find a unifier $\mathcal{C}(\sigma)$ of $\mathcal{C}(l \stackrel{?}{=} r)$ that satisfies the property

$\mathcal{C}(\sigma(l)) = \mathcal{C}(\sigma)(\mathcal{C}(l))$. This property is represented by the commutativity of the following diagram:

$$\begin{array}{ccc}
 l \stackrel{?}{=} r & \xrightarrow{\mathcal{C}} & \mathcal{C}(l \stackrel{?}{=} r) \\
 \downarrow \sigma & \xRightarrow{\mathcal{C}} & \downarrow \mathcal{C}(\sigma) \\
 \sigma(l) & \xrightarrow{\mathcal{C}} & \mathcal{C}(\sigma)(\mathcal{C}(l))
 \end{array}$$

Unfortunately, although we can curryify a unifier of a sequence unification problem Γ to obtain a unifier of the context unification problem $\mathcal{C}(\Gamma)$, the converse is not true: $f(V) \stackrel{?}{=} g(f)$ is trivially unsolvable, but its curry form, $X_V(a_f) \stackrel{?}{\approx} @ (a_g, a_f)$, is solvable: the substitution $[X_V \mapsto @(a_g, \bullet)]$ solves the context equation but there is no unifier for $f(V) \stackrel{?}{=} g(f)$. In general, solvability is not preserved by currying, i.e. the currying function is injective, but not surjective.

Example 1. The sequence unification problem

$$f(V_1, V_2) \stackrel{?}{=} f(f(a, V_2), f(V_2, a), b)$$

has these two unifiers:

$$\begin{aligned}
 \sigma_1 &= \{V_1 \mapsto \langle f(a), f(a), b \rangle, \quad V_2 \mapsto \langle \rangle\} \\
 \sigma_2 &= \{V_1 \mapsto \langle f(a, b), f(b, a) \rangle, \quad V_2 \mapsto \langle b \rangle\}
 \end{aligned}$$

When currying the problem we get the context unification problem:

$$X_{V_2}(X_{V_1}(a_f)) \stackrel{?}{\approx} @(@(@(a_f, X_{V_2}(@ (a_f, a_a))), @ (X_{V_2}(a_f), a_a)), a_b)$$

that has the following four solutions:

$$\begin{aligned}
 \varphi_1 &= \{X_{V_1} \mapsto @(@(@(\bullet, @ (a_f, a_a)), @ (a_f, a_a)), a_b), & X_{V_2} \mapsto \bullet\} \\
 \varphi_2 &= \{X_{V_1} \mapsto @(@(\bullet, @ (@ (a_f, a_a), a_b)), @ (@ (a_f, a_b), a_a)), & X_{V_2} \mapsto @(\bullet, a_b)\} \\
 \varphi_3 &= \{X_{V_1} \mapsto @(@(@ (a_f, @(\bullet, a_a)), @ (a_f, a_a)), a_b), & X_{V_2} \mapsto \bullet\} \\
 \varphi_4 &= \{X_{V_1} \mapsto @(@(@ (a_f, @ (a_f, a_a)), @(\bullet, a_a)), a_b), & X_{V_2} \mapsto \bullet\}
 \end{aligned}$$

It is easy to see that solutions φ_1 and φ_2 correspond respectively to σ_1 and σ_2 : $\varphi_1 = \mathcal{C}(\sigma_1)$ and $\varphi_2 = \mathcal{C}(\sigma_2)$, while φ_3 and φ_4 do not have any such “corresponding” solutions.

In the previous example, substitution for variable X_{V_1} in solutions φ_3 and φ_4 are not “well-typed”, i.e. they are not the curry form of any sequence of unranked terms. In φ_1 the variable X_{V_1} is mapped to the context $@(@(@(\bullet, @ (a_f, a_a)), @ (a_f, a_a)), a_b)$ that is the curry form of the sequence $\langle f(a), f(a), b \rangle$, whereas in φ_3 the variable X_{V_1} is mapped to the context $@(@(@ (a_f, @(\bullet, a_a)), @ (a_f, a_a)), a_b)$, that would be the curry form of something like $f(\langle a \rangle, f(a), b)$ which is not a sequence. In fact, \mathcal{C}^{-1} is not defined for $@(@(@ (a_f, @(\bullet, a_a)), @ (a_f, a_a)), a_b)$.

Thus, we can not assert that we can always reconstruct a unifier for the original problem from the unifier that we get for its curry form, we will need these unifiers to be well-typed.

Slightly abusing the notation, for a well-typed FC-substitution φ we denote by $\mathcal{C}^{-1}(\varphi)$ the IS-substitution defined as follows: $(\mathcal{C}^{-1}(\varphi))(w) = \mathcal{C}^{-1}(\varphi(\mathcal{C}(w)))$, for each $w \in \mathcal{V}$.

Lemma 3. *Let Γ be a sequence unification problem over Σ_{\cup} and \mathcal{V} , and let $\mathcal{C}(\Gamma)$ be its curried form. Assume φ is a well-typed (w.r.t Σ_{\cup}) unifier of $\mathcal{C}(\Gamma)$, then $\mathcal{C}^{-1}(\varphi)$ is a unifier of Γ .*

Proof: Let $\mathcal{C}(l) \stackrel{?}{\approx} \mathcal{C}(r) \in \mathcal{C}(\Gamma)$. Then $\varphi(\mathcal{C}(l)) = \varphi(\mathcal{C}(r))$. Since φ , $\mathcal{C}(l)$, and $\mathcal{C}(r)$ are well-typed, we get that $\varphi(\mathcal{C}(l))$ and $\varphi(\mathcal{C}(r))$ are well-typed as well. Therefore, $\mathcal{C}^{-1}(\varphi(\mathcal{C}(l)))$ and $\mathcal{C}^{-1}(\varphi(\mathcal{C}(r)))$ exist and $\mathcal{C}^{-1}(\varphi(\mathcal{C}(l))) = \mathcal{C}^{-1}(\varphi(\mathcal{C}(r)))$. From this, by definition of \mathcal{C}^{-1} for FC-substitutions we obtain $(\mathcal{C}^{-1}(\varphi))(l) = (\mathcal{C}^{-1}(\varphi))(r)$, i.e., $\mathcal{C}^{-1}(\varphi)$ is a unifier of $l \stackrel{?}{=} r \in \Gamma$. ■

Thus to preserve the set of solutions and ensure soundness in our transformation, i.e, to make the diagram commute, we can only consider well-typed unifiers. Now, we want to characterize these unifiers. As we have already argued in Remark 1, to be able to obtain a sequence from a context with \mathcal{C}^{-1} , the contexts must have a certain “shape”.

Definition 3. *A left-hole context is a context that has the hole in its leftmost position, i.e. that can be built with this grammar:*

$$L ::= \bullet \mid X(\bullet) \mid @(L, t)$$

for context variable X and hole-free ranked term t .

Lemma 4. *Let φ be a ground FC-substitution such that $\varphi(z) \in \mathcal{T}(\Sigma_{\cup}^{\mathcal{C}} \cup \{\bullet\}, \emptyset)$, for all $z \in \mathcal{V}^{\mathcal{C}}$. Then, φ is well-typed, iff $\varphi(X)$ is a left-hole context, for all context variable $X \in \mathcal{V}^{\mathcal{C}}$.*

Proof: By structural induction, from Definitions 2 and 1. ■

Now we define a variant of Context Unification, called Left-Hole Context Unification, as follows:

Definition 4. *Left-Hole Context Unification (LHCU) is a variant of Context Unification that requires instances of context variables to be left-hole contexts.*

Theorem 1. *Sequence Unification is P-reducible to Left-Hole Context Unification.*

Proof: The proof follows from Lemmas 2, 3 and 4. The \mathcal{C} function is polynomial in the sum of the sizes of the terms of the equations. ■

Hence, currying preserves solvability: Γ is a solvable SU problem, iff $\mathcal{C}(\Gamma)$ is a solvable LHCU problem. Moreover, from each unifier of a sequence unification problem we can reconstruct a unifier of the corresponding left-hole context unification problem, and from each *ground* left-hole context unifier we can get a unifier of the original sequence unification problem. Notice that some *non-ground* left-hole context substitutions, like $[X \mapsto @(•, @(y, a))]$, are not well-typed. The currying function is not onto, hence there are LHCU problems that are not the translation of any SU unification problem (see Section 5). Notice also that we assume that a LHCU problem is solvable, iff it has a *ground* left-hole context unifier. In fact, this is true, if we assume that the signature Σ_R contains at least a constant symbol.

4 Left-Hole Context Unification Decidability

In this section we reduce LHCU to Word Unification (WU) with Regular constraints, which is decidable [30]. Therefore, this reduction proves decidability of LHCU. The reduction is based on some ideas from [20]. There, it is proved (see Corollary 21) that if the rank-bound conjecture is true, then CU is decidable. The conjecture (see Conjecture 15) claims that there exists a computable upper bound for the Strahler number of some unifier of every solvable CU problem. Like in [20], the reduction will be done via the traversals of the terms that allows us to encode LHCU equations into WU equations. We need the regular constraints to make this encoding sound and ensure that the solutions of the WU equations really encode solutions of the corresponding LHCU problem. Here, we prove that there exists an upper bound for the rab and the Strahler numbers of minimal left-hole unifiers (see Lemmas 6 and 7).

In [21] it is proved that context unification is reducible to context unification with constants and only one binary symbol. The same reduction applies to LHCU. Therefore, from now on, we will assume that Σ_R only contains constants and a binary symbol that we represent as @. We also assume that Σ_R contains at least one constant. This is necessary to ensure that any solvable LHCU problem has a ground unifier. Moreover, we will also assume w.l.o.g. that we have just one initial context equation.

A naive encoding of a LHCU equation like $X(@(a, b)) \stackrel{?}{\approx} @(a, X(b))$ into a WU equation could be done using a postorder traversal of the terms of the equation as follows²:

$$\alpha_a \alpha_b \alpha_{@} W_X \stackrel{?}{\approx}_w \alpha_a \alpha_b W_X \alpha_{@}$$

where α_a, α_b and $\alpha_{@}$ are letters corresponding to a, b and @ respectively and W_X is the word variable that encodes the postorder traversal of the instantiation of the context variable X .

Then, some of the word solutions are:

$$\begin{aligned} \varphi_1 &= [W_X \mapsto \epsilon] \\ \varphi_2 &= [W_X \mapsto \alpha_{@}] \end{aligned}$$

² We use $\stackrel{?}{\approx}_w$ to denote word equations.

where ϵ is the empty word. Notice that $\varphi_2(W_X)$ does not correspond to a postorder traversal of a context, while $\varphi_1(W_X)$ is the postorder traversal of the empty context \bullet . This forces us to impose regular constraints to this encoding. In what follows we will show that with regular constraints we can get a sound encoding.

Definition 5. *Given a LHCU problem Δ , we say that φ is a minimal unifier, if there exists a most general unifier ρ such that $\varphi = [x_1 \mapsto a, \dots, x_n \mapsto a, X_1 \mapsto \bullet, \dots, X_m \mapsto \bullet] \circ \rho$, where $\{x_1, \dots, x_n, X_1, \dots, X_m\} = \text{vars}(\rho(\Delta))$ and a is a constant of Σ_R .*

Notice that any solvable LHCU problem has a minimal unifier. The following is an adaptation of the sound and complete set of rules for Linear Second-Order Unification [18] to LHCU. Its soundness and completeness proof can be adapted from [18].

Definition 6. *The unification procedure is described by a set of problem transformations, where every transformation has the form*

$$\langle \Delta \cup \{s \overset{?}{\approx} t\}, \varphi \rangle \Longrightarrow \langle \rho(\Delta \cup \Delta'), \rho \circ \varphi \rangle$$

and is characterized by a rule $s \overset{?}{\approx} t \Longrightarrow \Delta'$ and a substitution ρ .

Simplification: $a \overset{?}{\approx} a \Longrightarrow \emptyset$,
 $\text{@}(s_1, s_2) \overset{?}{\approx} \text{@}(t_1, t_2) \Longrightarrow \{s_1 \overset{?}{\approx} t_1, s_2 \overset{?}{\approx} t_2\}$,
 $x \overset{?}{\approx} x \Longrightarrow \emptyset$, and
 $X(s) \overset{?}{\approx} X(t) \Longrightarrow \{s \overset{?}{\approx} t\}$, where $\rho = []$ in the four cases.

Projection: $X(s) \overset{?}{\approx} t \Longrightarrow \{s \overset{?}{\approx} t\}$ and $\rho = [X \mapsto \bullet]$.

Imitation: $X(s) \overset{?}{\approx} \text{@}(t_1, t_2) \Longrightarrow \{X'(s) \overset{?}{\approx} t_1\}$ and $\rho = [X \mapsto \text{@}(X'(\bullet), t_2)]$,
 provided that X does not occur in t_2 ,³ and
 $x \overset{?}{\approx} s \Longrightarrow \emptyset$ and $\rho = [x \mapsto s]$, provided x does not occur in s .

Flex-Flex: $X(s) \overset{?}{\approx} Y(t) \Longrightarrow \{X'(s) \overset{?}{\approx} t\}$ and $\rho = [X \mapsto Y(X'(\bullet))]$,
 where $X \neq Y$.

The transformations are applied starting with $\langle \Delta, [] \rangle$ until we get a pair of the form $\langle \emptyset, \varphi \rangle$, or no transformation is applicable. In the first case, φ is a unifier of Δ , and, in the second case, the problem is unsolvable.

Proposition 1. *The unification procedure described in Definition 6 is sound: if $\langle \Delta, [] \rangle \Longrightarrow^* \langle \emptyset, \varphi \rangle$, then φ is a unifier of Δ , and complete: if φ is a most general unifier of Δ , then there exists a transformation sequence of the form $\langle \Delta, [] \rangle \Longrightarrow^* \langle \emptyset, \varphi \rangle$ ⁴*

³ The violation of these provisos leads to an occur-check error in the equations.

⁴ Notice that, for completeness, unifiers are required to be most general, but, in the soundness part, we can get non-most general unifiers.

Lemma 5. *Given a LHCU equation $s \stackrel{?}{\approx} t$, for any minimal unifier φ , and any context variable $X \in \text{vars}(s \stackrel{?}{\approx} t)$, we have $\varphi(X) = @(\dots @(\bullet, \varphi(t_n)) \dots, \varphi(t_1))$, where t_i is a subterm of s or of t , occurring as a second argument of an @, for all $1 \leq i \leq n$.*

Proof: From inspection of the transformation rules of Definition 6, we can see that right subterms (second arguments of @) are preserved: If $\langle \Delta_1, \varphi \rangle \implies^* \langle \Delta_2, \rho \circ \varphi \rangle$ and t is a subterm of Δ_2 occurring as a second argument of an @, then there exists a subterm t' in Δ_1 such that t' also occurs as a second argument of an @, and $t = \rho(t')$.

Now, by inspection of the transformation rules we can also see that the only transformation rule that introduces new right-subterms is the imitation rule, that introduces a right-subterm of the equations as a new right-subterm in the substitution. Therefore, if $\langle \Delta_1, \varphi \rangle \implies^* \langle \Delta_2, \rho \circ \varphi \rangle$, and t is a right-subterm of $\rho \circ \varphi(X)$, for some context variable X , then we can find an imitation step in the transformation sequence of one of the following forms:

$$\begin{aligned} \langle \Delta_1, \varphi \rangle &\implies^* \langle \Delta', \rho' \circ \varphi \rangle \implies \langle \Delta'', [Y \mapsto @(Y'(\bullet), s_2)] \circ \rho' \circ \varphi' \rangle \\ &\implies^* \langle \Delta_2, \rho'' \circ [Y \mapsto @(Y'(\bullet), s_2)] \circ \rho' \circ \varphi' \rangle \\ \langle \Delta_1, \varphi \rangle &\implies^* \langle \Delta', \rho' \circ \varphi \rangle \implies \langle \Delta'', [y \mapsto s] \circ \rho' \circ \varphi' \rangle \\ &\implies^* \langle \Delta_2, \rho'' \circ [y \mapsto s] \circ \rho' \circ \varphi' \rangle \end{aligned}$$

where either $X = Y$, $t = \rho''(s_2)$ and s_2 is a right-subterm of Δ' ; or Y [or y] is instantiated after X , and there is a right-subterm t' in s_2 [or s] such that $t = \rho''(t')$. Now, as we have proved, $s_2 = \rho'(t'')$ [or $t' = \rho'(t'')$], for some right-subterm t'' of Δ_1 . Therefore, $t = \rho(t'')$, for some right-subterm t'' of Δ_1 .

Completeness of the transformations ensure that any right-subterm of a most general unifier is an instance of a right-subterm of the original problem.

Finally, minimal unifiers can be obtained from most general unifiers, ensuring that instances of context variables have the form stated in the lemma. ■

The previous lemma allows us to prove that, if φ is a minimal unifier of $s \stackrel{?}{\approx} t$, then the number of times that we can go to the right descending through any branch of $\varphi(s)$, viewing the term as a tree, is bounded on the number of subterms of $s \stackrel{?}{\approx} t$.

Definition 7. *The number of right accumulated branches (rab) of a ground term $t \in \mathcal{T}^c$, noted $\text{rab}(t)$, is defined as:*

$$\begin{aligned} \text{rab}(a) &= 0 \\ \text{rab}(@ (t_1, t_2)) &= \max\{\text{rab}(t_1), 1 + \text{rab}(t_2)\} \end{aligned}$$

Lemma 6. *Let $s \stackrel{?}{\approx} t$ be a LHCU equation and φ a minimal unifier, then $\text{rab}(\varphi(s)) \leq |s| + |t|$.*

Proof: Lemma 5 ensures that, if φ is a minimal unifier, then for any subterm t_1 of $\varphi(s)$ occurring as a second-argument of an @ there exists a subterm t_2 in $s \stackrel{?}{\approx} t$ such that $t_1 = \varphi(t_2)$. Therefore, since there are $|s| + |t|$ subterms in $s \stackrel{?}{\approx} t$, and we can not repeat the same subterm in a branch, $\text{rab}(\varphi(s)) \leq |s| + |t|$. ■

The definition of rab is similar to the definition of the Strahler number of a term:

Definition 8. *The Strahler Number of a term t built up from binary and nullary symbols, noted $\text{Strahler}(t)$, is defined recursively as follows:*

$$\begin{aligned} \text{Strahler}(a) &= 0 \\ \text{Strahler}(@ (t_1, t_2)) &= \begin{cases} \text{Strahler}(t_1) + 1 & \text{if } \text{Strahler}(t_1) = \text{Strahler}(t_2) \\ \max\{\text{Strahler}(t_1), \text{Strahler}(t_2)\} & \text{otherwise} \end{cases} \end{aligned}$$

for any constant a , and the binary symbol $@$.

Since, we have $\text{Strahler}(t) \leq \text{rab}(t)$, for any term t , we can prove the following.

Lemma 7. *Given an LHCU equation $s \stackrel{?}{\approx} t$, all minimal unifiers φ satisfy $\text{Strahler}(\varphi(t)) \leq |s| + |t|$.*

The previous lemma proves the rank-bound conjecture of [20] for a variant of context unification. Therefore, we can conclude decidability of LHCU from a small modification of the results of that paper. That proof was based on the use of traversals of terms, and on *traversal equations*. These traversal equations were reduced to word equations with regular constraints. Here, we find an easier way to constraint traversals of $\varphi(s)$ with regular expressions. These regular expressions define postorder traversals of terms with a bounded rab and allows us to avoid the use of traversal equations which can be replaced by simple word equations with regular constraints. What follows is then an alternative proof for the decidability of LHCU based on some ideas of [20].

Definition 9. *Let Σ_0 be the set containing a distinct letter α_a , for every constant $a \in \Sigma_R$, and let $\alpha_@$ be also a letter (corresponding to the function symbol $@$)⁵ Let us consider the following family of regular languages*

$$\begin{aligned} L^0 &= \Sigma_0 \\ L^1 &= \Sigma_0 (L^0 \alpha_@)^* \\ &\vdots \\ L^n &= \Sigma_0 (L^{n-1} \alpha_@)^* \end{aligned}$$

Lemma 8. *The language L^n defines the set of postorder traversals of ground terms $t \in \mathcal{T}(\Sigma_R, \emptyset)$ satisfying $\text{rab}(t) \leq n$.*

Theorem 2. *LHCU can be reduced to WU with regular constraints.*

Proof: Assume that, apart from $\alpha_@$ and from a distinct letter α_a , for every function symbol $a \in \Sigma_R$, we also have a distinct word variable W_z , for every context or first-order variable $z \in \mathcal{X}$. The reduction uses the following transformation:

$$\begin{aligned} \mathcal{R}(a) &= \alpha_a \\ \mathcal{R}(@ (t_1, t_2)) &= \mathcal{R}(t_1) \mathcal{R}(t_2) \alpha_@ \\ \mathcal{R}(x) &= W_x \\ \mathcal{R}(X(t)) &= \mathcal{R}(t)W_X \end{aligned}$$

⁵ Notice that from previous assumptions $\Sigma_R \neq \emptyset$.

The translation is extended to context equations as $\mathcal{R}(s \stackrel{?}{\approx} t) = \mathcal{R}(s) \stackrel{?}{=} \mathcal{R}(t)$. We have to add the following regular constraints $W_x \in L^n$, for every first-order variable x , and $\alpha_a W_X \in L^n$, for every context variable X , where a is any of the constants of Σ_R , and $n = |s| + |t|$.

Extending the translation to substitutions, like it is done in the Section 3, we can prove easily that the translation maps minimal context unifiers to word unifiers. To prove that word unifiers may be decoded into context unifiers we need to use the regular constraints and Lemma 8. ■

Corollary 1. *Left-Hole Context Unification is decidable.*

5 Back to the Beginning

Now we look back at where we started from: Sequence Unification. Decidability of LHCU proved in the previous section gives another decidability proof of SU. Looking at the proof closer, we notice that we prove something more: Decidability of unification for an extension of SU. This extension, denoted ESU, is obtained if we allow individual variables to occur in functional positions, and a term to be applied to a sequence of terms. This is motivated by the fact that LHCU problems may contain terms like, e.g., $@(x_v, a)$ that could be obtained if we had currying defined for $v(a)$. We do not go into formal details here because of space limitation. The following example can serve for illustrating ESU:

Example 2. Extended sequence unification problem $\{f(a, V) \stackrel{?}{=} v(a, b)\}$ has two mgu's: $\sigma_1 = \{V \mapsto b, v \mapsto f()\}$ and $\sigma_2 = \{V \mapsto \langle U, a, b \rangle, v \mapsto f(a, U)\}$. Applying σ_2 to $v(a, b)$ gives $f(a, U, a, b)$.

Decidability of ESU can be shown based on decidability of LHCU. The sequence unification procedure 16 can be easily adapted to obtain a minimal complete unification procedure for ESU.

Moreover, we can transfer some of the results on complexity of Context Matching 29 to Extended Sequence Matching (ESM). The counterparts of linear context matching and arity 2 context matching problems are linear ESM (LESM) and arity 2 ESM (V2ESM), respectively. Shared-linear context matching gives a fragment of ESM that we call *prefix-closed ESM* (PCESM). It can be characterized by the following property: If a sequence variable V occurs in the subterms $f_1(r_1, \dots, r_n, V, \dots)$ and $f_2(l_1, \dots, l_m, V, \dots)$, where $f_1, f_2 \in \Sigma_U \cup \mathcal{V}$, then $f_1 = f_2$, $n = m$, and $r_i = l_i$ for each $1 \leq i \leq n$. It means that prefixes of all occurrences of a sequence variable should be the same. Then we have the following theorem, that follows from the analogous results in 29 and the construction of curry function:

Theorem 3. *LESM and PCESM are in P. V2ESM is NP-complete.*

It is hard to characterize a fragment of Sequence Matching obtained by inverse currying from Stratified Context matching. There is no obvious pattern in the form of such sequence matching problems.

6 Conclusion

We study the relation between two generalizations of Word Unification: Sequence Unification and Context Unification. We introduce a transformation function to translate sequence unification problems into context unification problems over a signature with constants and a single binary function symbol. The transformation preserves solvability in one direction: from SU to CU. To preserve solvability in the other direction, we add a restriction on the form of solutions of context unification problems, obtaining the left-hole variant of CU. We prove that a sequence unification problem is solvable iff the corresponding left-hole context unification problem is solvable, and the unifiers can be reconstructed in both directions. Moreover, we prove that LHCU is decidable, reducing it to WU with regular constraints. This result gives a decidability proof for an extension of SU, and, in particular, a new proof of decidability of SU. Based on the transformation, we transfer some complexity results from context matching to sequence matching.

References

1. Boley, H.: A Tight, Practical Integration of Relations and Functions. LNAI vol. 1712, Springer, Heidelberg (1999)
2. Buchberger, B., Crăciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards computer-aided mathematical theory exploration. *J. Applied Logic* 4, 470–504 (2006)
3. Chasseur, E., Deville, Y.: Logic program schemas, constraints and semi-unification. In: Fuchs, N.E. (ed.) LOPSTR 1997. LNCS, vol. 1463, pp. 69–89. Springer, Heidelberg (1998)
4. Coelho, J., Florido, M.: CLP(Flex): Constraint logic programming applied to XML processing. In: Meersman, R., Tari, Z. (eds.) On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. LNCS, vol. 3291, pp. 1098–1112. Springer, Heidelberg (2004)
5. Coelho, J., Florido, M.: VeriFLog: A constraint logic programming approach to verification of website content. In: Shen, H.T., Li, J., Li, M., Ni, J., Wang, W. (eds.) Advanced Web and Network Technologies, and Applications. LNCS, vol. 3842, pp. 148–156. Springer, Heidelberg (2006)
6. Common Logic Working Group. Common Logic Standard (2007)
<http://philebus.tamu.edu/cl/>
7. Diekert, V.: Makanin’s algorithm. In: Algebraic aspects of combinatorics on words, chapter 12, pp. 342–390. Cambridge University Press, Cambridge (2002)
8. Genesereth, M.R., Petrie, C., Hinrichs, T., Hondroulis, A., Kassoff, M., Love, N., Mohsin, W.: Knowledge Interchange Format, draft proposed American National Standard (dpANS) Technical Report NCITS.T2/98-004 (1998)
9. Ginsberg, M.L.: The MVL theorem proving system. *SIGART Bull.* 2(3), 57–60 (1991)
10. Hamana, M.: Term rewriting with sequences. In: Proc. of the First Int. Theorema Workshop. Technical report 97–20, RISC, Linz, Austria (1997)

11. Hayes, P., Menzel, C.: Semantics of Knowledge Interchange Format (2001) <http://reliant.teknowledge.com/IJCAI01/HayesMenzel-SKIF-IJCAI2001.pdf>
12. Hayes, P.J., Menzel, C.: Simple common logic. In: W3C Workshop on Rule Languages for Interoperability. W3C (2005)
13. Jaffar, J.: Minimal and complete word unification. *J. ACM* 37(1), 47–85 (1990)
14. Koller, A.: Evaluating context unification for semantic underspecification. In: 3rd ESSLLI Student Session (ESSLLI'98), August 17–28, pp. 188–199 (1998)
15. Kutsia, T.: Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In: Calmet, J., Benhamou, B., Caprotti, O., Henocque, L., Sorge, V. (eds.) AISC 2002 and Calculemus 2002. LNCS (LNAI), vol. 2385, pp. 290–304. Springer, Heidelberg (2002)
16. Kutsia, T.: Solving equations with sequence variables and sequence functions. *Journal of Symbolic Computation* 42(3), 352–388 (2007)
17. Kutsia, T., Marin, M.: Matching with regular constraints. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 215–229. Springer, Heidelberg (2005)
18. Levy, J.: Linear second-order unification. In: Ganzinger, H. (ed.) Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA'96). LNCS, vol. 1103, pp. 332–346. Springer, Heidelberg (1996)
19. Levy, J., Niehren, J., Villaret, M.: Well-nested context unification. In: Nieuwenhuis, R. (ed.) Proc. of the 20th Int. Conf. on Automated Deduction, CADE-20. LNCS (LNAI), vol. 3632, pp. 149–163. Springer, Heidelberg (2005)
20. J. Levy and M. Villaret. Context unification and traversal equations. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA'01)*, volume 2041 of LNCS, pages 169–184, 2001.
21. Levy, J., Villaret, M.: Curryng second-order unification problems. In: Tison, S. (ed.) RTA 2002. LNCS, vol. 2378, pp. 326–339. Springer, Heidelberg (2002)
22. Makanin, G.S.: The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik* 32(2), 129–198 (1977)
23. Marin, M., Kutsia, T.: Foundations of the rule-based system RhoLog. *Journal of Applied Non.-Classical Logics* 16(1–2), 151–168 (2006)
24. Niehren, J., Pinkal, M., Ruhrberg, P.: A uniform approach to underspecification and parallelism. In: Proc. of the 35th Annual Meeting of the ACL and the 8th Conf. of the European Chapter of the ACL (ACL'97), pp. 410–417 (1997)
25. Paulson, L.: Isabelle: the next 700 theorem provers. In: Odifreddi, P. (ed.) Logic and Computer Science, pp. 361–386. Academic Press, San Diego (1990)
26. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: Proc. of the 40th Annual Symp. on Foundations of Computer Science, FOCS'99, New York City, USA, pp. 495–500. IEEE Press, Orlando, Florida, USA (1999)
27. Richardson, J., Fuchs, N.E.: Development of correct transformation schemata for Prolog programs. In: Fuchs, N.E. (ed.) LOPSTR 1997. LNCS, vol. 1463, pp. 263–281. Springer, Heidelberg (1997)
28. Schmidt-Schauß, M.: A decision algorithm for stratified context unification. *Journal of Logic and Computation* 12, 929–953 (2002)
29. Schmidt-Schauß, M., Stuber, J.: The complexity of linear and stratified context matching problems. *Theory of Computing Syst.* 37(6), 717–740 (2004)
30. Schulz, K.U.: Makanin's algorithm for word equations – two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992)
31. Wolfram, S.: The Mathematica Book. Wolfram Media, 5th edn. (2003)

The Termination Competition

Claude Marché^{1,2} and Hans Zantema³

¹ INRIA Futurs, ProVal, Parc Orsay Université, F-91893

² Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

³ Department of Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

Claude.Marche@inria.fr, h.zantema@tue.nl

Abstract. Since 2004, a Termination Competition is organized every year. This competition boosted a lot the development of automatic termination tools, but also the design of new techniques for proving termination. We present the background, results, and conclusions of the three first editions, and discuss perspectives and challenges for the future.

1 Motivation and History

In a landmark paper in 1970, Manna & Ness [1] proposed a criterion for proving termination of rewrite systems, based on *reduction* orderings. Since then, many techniques for proving termination have been proposed, by providing means of defining classes of reduction orderings: path orderings, polynomial interpretations, etc. A few implementations were developed [2,3,4] but practical results in proposed sets of benchmark problems [5,6] were quite poor.

A disruptive progress came up in 1997, with the *Dependency Pair* criteria proposed by Arts & Giesl [7], allowing to prove termination with a larger class of orderings than reduction orderings. This brought up a new interest towards automation of termination proofs. Since around 2000 several tools were developed for this goal, and in 2003, for the Workshop on Termination in Valencia, Albert Rubio organized a special session for comparing tools. Tool authors gathered, proposed a few challenging examples, and each of them manually ran their own tool and told what they were able to prove. Participants were AProVE [8], Cariboo [9], CiME [10], MatchBox [11], Temptation [12] and TTT [13] on rewrite systems problems; and TALP [14], TerminWeb [15] and Hasta-La-Vista [16] for logic programs. MatchBox was only dealing with string rewriting.

Stimulated by the enthusiasm of the participants it was decided to organize an annual competition. Participants agreed on a common syntax of problems, in order to build a shared database called the TPDB (Termination Problem Data Base, <http://www.lri.fr/~marche/tpdb/>). The main idea was that the competition must be run fully automatically, to demonstrate the ability of tools to solve termination problems, without requiring any expertise use, such as setting clever options or parameters as what the author's tool can do. C. Marché took care of the organization and the development of the required utilities for running the competition automatically and making results available online. The main objectives for such a competition were and remain:

- to stimulate research in this area, shifting emphasis towards automation,
- to provide a standard to compare termination techniques.

The first full competition in this style ran in May 2004, the week before the Workshop on Termination in Aachen, where the results were reported. There were three categories, corresponding to different input syntax: term rewriting (TRS, 5 participating tools), string rewriting (SRS, 5 tools) and logic programs (LP, 2 tools). With respect to the first event in 2003, there was only one new tool: TORPA [17], specialized to SRSs. Some other tools were not able to participate in this automatic competition: *Termptation*, *TerminWeb* and *Hasta-La-Vista*. *AProVE* had a new module allowing it to participate to the LP category. The TRS category has been subdivided into 5 sub-categories corresponding to standard rewriting, rewriting modulo theories, innermost strategy, context-sensitive strategy and conditional rewriting.

In 2005, Hans Zantema joined the organization of the competition, and the second competition ran in April 2005, the week before RTA in Nara. This time there were only two categories: TRS and SRS. In TRS category, *Cariboo* did not participate but 2 new tools joined: TPA [18] and TEPARLA [19], leading to a total of 6 tools. They also participated to the SRS category, together with the new tool *JamBox* [20], thus there were 8 participants. A new sub-category for relative termination was introduced, both for term rewriting and string rewriting. In the meantime, the size of the TPBD grew significantly. Several termination proofs were given for systems where the 2004 versions failed, showing improvements of the tools. The ranking of the tools were quite similar as the 2004 edition.

In 2006, the competition was held in June, two months before WST and RTA in Seattle. In the TRS category, there were eight participants: TTT was replaced by a variation called TTTbox [21], and two new tools joined: *MU-Term* [22] and *JamBox* [20] (which was only in SRS category in the previous year). Nine tools were in the SRS category: a new tool *MultumNonMultum* [23] joined. This year, the rankings have been significantly modified.

In the following, we first describe in Section 2 the rules of the competition. Then in Section 3 we summarize the results and comment their evolution over the years. We draw some conclusions and perspectives in Section 4 and we provide a list of challenges for future competitions in Section 5.

2 The Rules

The following rules were applied to the 2006 termination competition.

- Submission of new problems for TPDB is open until a few weeks before the competition, when this new TPDB is publicly available for testing and tuning the tools.
- Just before the competition participants submit
 - final versions of the tools, and
 - secret problems, up to ten per participant per category, that are added to the version of TPDB used for the competition, but not accessible for other participants before the competition.

- All tools apply on all problems in the corresponding TPDB categories, all on the same machine. The required output of every tool is
 - “YES”, followed by the text of a termination proof, or
 - “NO”, followed by the text of a non-termination proof, or
 - anything else, interpreted as “DON’T KNOW”.
- Execution of more than one minute for any tool on any termination problem causes a time-out, interpreted as “DON’T KNOW”.
- All results are reported on-line, including generated proof text, and statistics about scores and running time.
- Any tool generating a wrong answer is “disqualified” (see Section 3.4).
- There are no formal rules and consequences of being a “winner”, apart from the honour of having a high or the highest score in some (sub)category.

These rules were designed in such a way that participants also being organizer had no advantage of being organizer. In 2006, categories were subdivided in the following eight subcategories:

- Standard term rewriting.
- Innermost term rewriting. This means that only rewrite steps are allowed for which all proper subterms of the redex are in normal form.
- Context-sensitive strategy. This means that for every operation symbol it is specified under which position rewriting is allowed.
- Term rewriting modulo theory. This means that apart from the rewrite rules also equations are specified (usually associativity and commutativity) modulo which rewriting is done.
- Relative termination of term rewriting. This means that two rewrite systems R, S are specified for which termination of $\rightarrow_S^* \cdot \rightarrow_R \cdot \rightarrow_S^*$ has to be proved.
- Standard string rewriting. This coincides with standard term rewriting in which all symbols have arity one.
- Relative termination of string rewriting.
- Logic programs.

3 Competitions Results

We present here a summary of the results in the three first editions of the competition. We only present and discuss on the two main categories: standard string rewriting and standard term rewriting. More detailed results including all termination problems, all generated proofs, executable code of the tools, and measured execution times and statistics are available from <http://www.lri.fr/~marche/termination-competition/>.

3.1 SRS Category

The following table summarizes the results of the SRS category, for the three editions of the competition. In the second column we give the number of problems submitted to tools. Then for both YES and NO answers, we give the total number

of problems solved, then the three best tools with their respective score with respect to this total. We also give the number of problems that remain unsolved by any tool.

year	# pbs	answer	overall	1st		2nd		3rd	
2004	104	yes	89	TORPA	88	AProVE	87	MatchBox	49
		no	7	MatchBox	7	TORPA	6	AProVE	5
		unk.	8						
2005	153	yes	139	TORPA	126	AProVE	114	JamBox	102
		no	9	JamBox and MatchBox		9	TORPA	5	
		unk.	5						
2006	322	yes	263	JamBox	251	TORPA	201	MatchBox	176
		no	25	JamBox	25	AProVE and MatchBox		12	
		unk.	34						

After its short win in 2004, TORPA has been a clear winner in 2005. But a big surprise happened in 2006: the new JamBox tool was significantly improved and became clearly more efficient than TORPA. The reason is the new technique of matrix interpretations implemented in JamBox, discussed below. It is also noticeable that the number of problems significantly grew in 2006, incorporating sets of problems from new research groups. This clearly made the competition exciting, and the next competition will be interesting to follow.

3.2 TRS Category

year	# pbs	answer	overall	1st		2nd		3rd	
2004	521	yes	426	AProVE	410	TTT	397	CiME	297
		no	22	AProVE	22	MatchBox	21	TTT	12
		unk.	73						
2005	773	yes	588	AProVE	576	TTT	509	TPA	407
		no	94	AProVE	94	MatchBox	80	TEPARLA	15
		unk.	91						
2006	865	yes	686	AProVE	638	JamBox	626	TPA	422
		no	103	AProVE	103	MatchBox	85	TEPARLA	15
		unk.	76						

The clearly visible fact is that the AProVE tool has been constantly the best tool each year, both for proving and disproving termination. However, this should not hide that this required strong improvement each year: for example, the AProVE 2005 version would not have won the 2006 edition. Indeed, the JamBox tool made the 2006 competition very exciting, showing how efficient was the new technique of matrix interpretation implemented in JamBox. It is noticeable too that each year, there were a significant amount of problems not solved by the winner but solved by others.

Finally, notice that although the number of problems increased by 92 between 2005 and 2006, the number of undecided problems decreased by 15: this is a clear evidence that the tools efficiency considerably improved.

3.3 Other Categories

To summarize briefly the other categories: AProVE remained winner in every sub-category, except relative termination which it does not support. JamBox won relative termination subcategories for TRS and SRS in 2006.

In the Logic Programs category, AProVE won both 2004 and 2006 editions. The evolution in this category is poorly significant, it seems that the lack of participants does not encourage efforts in this direction. In other words, the competition cannot reach its goal of stimulating research if existing other tools are not willing to participate.

3.4 Remarks

soundness issue Since all tools execute complicated tasks it is likely that they contain bugs. In 2006, for two tools (CIME and MU-Term) we detected some obviously incorrect generated proofs. The tools have been “disqualified” in the sense that their scores are not taken into account (results above would not be different anyway). We emphasize that it does not imply that all termination proofs generated by the remaining tools are correct: we cannot check all thousands of generated termination proofs. As a long-term objective we see an automatic formal correctness check of the generated proofs.

timing issue Most proofs are found within less than a second. For most tools the average time to find a termination proof was a few seconds. In 2006, we experimented running a second round for problems not solved in a one minute timeout, giving a time limit of five minutes instead. In the category of term rewriting it occurred a few times that a termination proof was found by a tool in the second round where all tools failed in the first round: 3 times for standard rewriting and once for context-sensitive and modulo theory subcategories. In the string rewriting category, the tools JamBox, MultumNonMultum and TPA found termination proofs in a second round where all tools failed in the first round. The total number of these systems was 5, both in the subcategories standard (2) and relative termination (3). The time limit has a small influence on the set of unsolved problems, emphasizing the fact that it is not easy to know when a tool does not solve a problem, whether it is just because of lack of time, or because of an intrinsic insufficiency of the techniques implemented.

non-termination Since the first competition in 2004, it was decided to evaluate the ability of proving non-termination. Before that, no tool except maybe MatchBox was implementing any technique for disproving termination. We emphasize that the competition stimulated both the development of new techniques for disproving termination, and of course implementations (but still not all tools have facilities for this). The non-termination proofs found were all generated by presenting a looping reduction. For logic programs, no tool is able to give non-termination proofs. For the rewriting categories, only very few proofs of non-termination are obtained for other sub-categories than standard rewriting.

challenging problems After the 2004 competition, we tried to designate a subset of unsolved problems that could represent challenges. We chose some modifications of Toyama's system, and they were solved in the 2005 competition. Again, in 2005 we presented termination of the SRS `Zantema-z086` consisting of the three rules $aa \rightarrow bc$, $bb \rightarrow ac$, $cc \rightarrow ab$ as an open problem, since no tool solved this system in the two first editions. It was also proposed as a new problem in the RTA open problem list. This has been first solved by Hofbauer and Waldmann [24], and without any doubt it stimulated the development of the new matrix interpretation method [25,26], by which the problem was automatically proved terminating by JamBox in 2006.

4 Conclusions and Perspectives

The three first edition of the Termination Competition have been really exciting due to new developments of termination techniques and new implementations. Dependency graph criteria [7], together with efficient techniques to search for argument filterings, path orderings and polynomial interpretations, were shown to be the most useful techniques. The most powerful new technique whose discovery was motivated by the competition is the matrix method [25,26]. Regarding implementations of this technique, we emphasize that JamBox and MatchBox both use an external SAT solver for searching for suitable interpretations. AProVE also uses a SAT solver to search for path orderings and polynomial interpretations [27]. It seems to be a very efficient way to benefit from the very good progress made by SAT solvers, which maybe indeed due to the existing competition (<http://www.satcompetition.org/>) of this kind.

We also emphasize that although AProVE won all three competitions in standard TRS category, it couldn't remained winner each year without major improvements in the techniques implemented: handling applicative TRSs [28], polynomials with negative coefficients [29], subterm criterion [30], match-bounds for term rewriting [31], etc.

We consider that the termination competition has been very successful so far, justifying annual continuation:

- It provides an objective way to compare the power of various implementations and techniques for proving termination.
- New challenges emerge from the competition, stimulating the development of new powerful techniques.

As remarked in Section 3.4, an important objective for the future is an automatic formal correctness check of generated proofs. Achieving this both requires a lot of work and agreement about formats of the proofs. But recent progress has been made by two research groups using the Coq proof assistant for formalizing a soundness proof checker: CoLoR [32] and A3PAT/CiME [33]. It is likely that a sub-category for certified termination proof will appear in the next competition.

The emphasis in the competition is in rewriting rather than termination of programs. For logic programs, even if recent progresses have been made, the

participating tools restricted to the specific technique of transforming the logic program to a term rewriting system and then prove termination of the latter. We should like to have participation by other tools not focusing on rewriting, such as the TERMINATOR tool [34]. For the next competition we plan to add new categories for functional programs (Haskell, ML) and for some kind of imperative programs. Another aspect, which was forgotten when focusing on rewrite systems, is the support for numerical computations, using for example built-in integers. This was indeed an important originality of the Hasta-La-Vista tool for logic programs. This aspect should be considered again in the future, to support such numerical computations in functional or imperative programs.

5 Challenges

To stimulate further research, we now present some challenging problems. All references in typewriter font refer to TBDB. Right after the end of the 2006 competition, it was emphasized that the SRS `SRS/Waldmann-jw1`

$$aaa \rightarrow bab \qquad bbb \rightarrow aaa$$

was probably the shortest TPDB problem unsolved by any tool. In the meantime, it has been solved by A. Nogin and C. Witty (<http://lists.lri.fr/pipermail/termtools/2006-August/000295.html>) and now can be solved automatically [35].

5.1 Longstanding Open Problems

Challenge 1 (Hercules and Hydra battle). *It is problem TRS/D33-33, the famous Hercules and Hydra battle. It is the only problem of old benchmarks [5,6] which remain unsolved by any tool.*

$$\begin{array}{ll} h(z, e(x)) \rightarrow h(c(z), d(z, x)) & d(z, g(0, 0)) \rightarrow e(0) \\ d(z, g(x, y)) \rightarrow g(e(x), d(z, y)) & g(e(x), e(y)) \rightarrow e(g(x, y)) \\ d(c(z), g(g(x, y), 0)) \rightarrow g(d(c(z), g(x, y)), d(z, g(x, y))) & \end{array}$$

At several occasions, some tool author pretended to be able to solve it, but it never happened during the competition, and it is impossible to know whether it was requiring specific user interaction, or even if it was not a bug in the tools.

Several other TRSs have been proposed in the literature, for various purposes, for which the termination was left as an open problem. In 1997, a TRS for integer arithmetic [36] was presented, with a very complicated ad-hoc proof. It has been shown terminating using MSPO in 2003, and in the same year solved automatically by CiME using dependency pairs modulo AC and polynomial interpretations. In the same article, a large TRS for rational arithmetic is presented, whose termination remains open. In 2000, Deplagne [37] introduced a TRS for sequent calculus modulo, leaving again open its termination. It is now shown

terminating both by AProVE and CiME. Bonelli proposed a TRS for explicit substitutions, and was unable to prove its termination. It has been introduced as a secret problem in the 2005 competition (TRS/secret2005-cime1), and was first shown to be terminating by TEPARLA. Another system related to explicit substitution is TRS/Zantema-z10. Several papers were devoted only to proving termination of this system; a shorter proof based on semantic labelling [38] was first found manually, and later on by TPA.

A very long-standing open termination problem appeared in 1991: a TRS proposed by Cohen and Watson [39] for arithmetic:

Challenge 2 (Cohen-Watson system for arithmetic). *This is the 22-rules system TRS/secret2006-cime1 of the TPDB. Its termination status is unknown, and is indeed the problem #65 in the RTA List of Open Problems.*

5.2 Semantic Decreasing Argument

A large class of problems which remain unsolved in the TPDB seem to require a decreasingness argument that is “semantic”: it depends on a value of a normal form of a subterm (which is more or less a return value of some auxiliary function call). This is especially true for TRSs coming from automatic translations of context-sensitive TRSs, or Maude or OBJ programs. This phenomenon may occur also when dealing with functional or imperative programs.

Challenge 3 (While loop). *As an instance of this problem, we mention TRS/Zantema06-while:*

$$f(t, x, y) \rightarrow f(g(x, y), x, s(y)) \quad g(s(x), 0) \rightarrow t \quad g(s(x), s(y)) \rightarrow g(x, y)$$

which encodes the obviously terminating loop while $x > y$ do $y := y + 1$. Using the semantics, e.g., by semantic labelling [38], a termination proof can be given, but until now no tool could solve it.

5.3 String Rewriting Systems

In 2006, three series of randomly generated SRSs were added, and those problems are poorly solved by tools, hence those series provide a set of challenging problems. We propose below a challenging problem with a more natural computational content.

Challenge 4 (Power 2 to power 3). *This is problem SRS/Zantema-z079:*

$$caa \rightarrow ac \quad acb \rightarrow adb \quad ad \rightarrow daaa \quad bd \rightarrow bc$$

which essentially rewrites 2^n to 3^n , more precisely $bca^{2^n}b$ rewrites to $bca^{3^n}b$. No tool could solve it, for any of the three editions of the competition.

5.4 Non-termination

There are also challenging problems for showing non-termination. An example of these is TRS/HofWald-6:

$$f(f(a, x), y) \rightarrow f(f(x, f(a, y)), a)$$

where a is a constant and x, y are variables, which is not solved by any tool, although there seems to be a very simple and easy implementable way to observe non-termination: every ground instance of the right hand side contains an instance of the left hand side. Another example is the following SRS,

$$al \rightarrow la \quad ra \rightarrow ar \quad bl \rightarrow bar \quad rb \rightarrow lb$$

being SRS/Zantema-z073. It is obviously non-terminating since $ba^n lb \rightarrow^* ba^{n+1} lb$ for all n , but no tool can solve it. Generally speaking, techniques should be developed for automatically proving non-looping non-termination.

Acknowledgements. Special thanks to Albert Rubio who brought up the idea of a termination competition, and set up the first TPDB. Thanks to all participants all over the years.

References

1. Manna, Z., Ness, S.: On the termination of Markov algorithms. In: ICSS. pp. 789–792 (1970)
http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html
2. Lescanne, P.: Computer experiments with the REVE term rewriting system generator. In: Proc. POPL'83 (1983)
3. Forgaard, R., Detlefs, D.: Reve 2.4: A program for generating and analyzing term rewriting systems. Massachusetts Institute (1984)
4. Steinbach, J.: Generating polynomial orderings. Information Processing Letters 49, 85–93 (1994)
5. Dershowitz, N.: 33 examples of termination. In: Comon, H., Jouannaud, J.-P. (eds.) Spring School of Theoretical Computer Science. LNCS, vol. 909, pp. 16–26. Springer, Heidelberg (1995)
6. Steinbach, J., Kühler, U.: Check your ordering – termination proofs and open problems. Technical Report SEKI Report SR-90-25, Univ. Kaiserslautern (1990)
7. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
8. Giesl, J., Schneider-Kamp, P., Thiemann, R.: The AProVE tool (RWTH, Aachen, Germany) <http://aprove.informatik.rwth-aachen.de/>
9. Fissore, O., Gnaedig, I., Kirchner, H.: CARIBOO: A multi-strategy termination proof tool based on induction. In: Rubio, A., ed.: WST, pp. 77–79 (2003)
10. Contejean, E., Marché, C., Urbain, X.: The CiME rewrite toolbox (LRI, Orsay, France) <http://cime.lri.fr>
11. Waldmann, J.: The MatchBox tool (HTWK, Leipzig, Germany)
<http://dfa.imn.htwk-leipzig.de/matchbox/>

12. Borralleras, C., Rubio, A.: The termptation tool (Universitat Politecnica de Catalunya, Spain) <http://www.lsi.upc.es/~albert/term.html>
13. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool (Innsbruck Universität, Austria) <http://colo6-c703.uibk.ac.at/ttt/>
14. Claves, C., Ohlebusch, E.: The TALP tool (University of Bielefeld, Germany) <http://bibiserv.techfak.uni-bielefeld.de/talp/>
15. Codish, M., Taboch, C.: the TerminWeb tool (Ben Gurion University, Israel) <http://www.cs.bgu.ac.il/~mcodish/TerminWeb/>
16. Serebrenik, A., De Schreye, D.: The Hasta-La-Vista tool (K.U. Leuven, Belgium)
17. Zantema, H.: The TORPA tool (Technische Universiteit Eindhoven, The Netherlands) <http://www.win.tue.nl/~hzantema/torpa.html>
18. Koprowski, A.: The TPA tool (Technische Universiteit Eindhoven, The Netherlands) <http://www.win.tue.nl/tpa>
19. van der Wulp, J.: The TEPARLA tool (Technische Universiteit Eindhoven, The Netherlands) <http://www.win.tue.nl/~hzantema/torpa.html>
20. Endrullis, J.: The JamBox tool (Free University in Amsterdam, The Netherlands) <http://joerg.endrullis.de/>
21. Korp, M.: The TTTbox (Innsbruck Universität, Austria) <http://homepage.uibk.ac.at/~csad2836/TTTbox.html>
22. Lucas, S.: Mu-term, a tool for proving termination of rewriting with replacement restrictions (2003) <http://www.dsic.upv.es/~slucas/csr/termination/muterm/>
23. Hofbauer, D.: The MultumNonMultum tool (Kassel, Germany) <http://www.theory.informatik.uni-kassel.de/~dieter/multum/>
24. Hofbauer, D., Waldmann, J.: Termination of $\{aa \rightarrow bc, bb \rightarrow ac, cc \rightarrow ab\}$. Information Processing Letters 98(4), 156–158 (2006)
25. Hofbauer, D., Waldmann, J.: Termination of string rewriting with matrix interpretations. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, Springer, Heidelberg (2006)
26. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, Springer, Heidelberg (2006)
27. Codish, M., Schneider-Kamp, P., Lagoon, V., Thiemann, R., Giesl, J.: Sat solving for argument filterings. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 30–44. Springer, Heidelberg (2006)
28. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCoS. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
29. Hirokawa, N., Middeldorp, A.: Polynomial interpretations with negative coefficients. In: Buchberger, B., Campbell, J.A. (eds.) AISC 2004. LNCS (LNAI), vol. 3249, pp. 185–198. Springer, Heidelberg (2004)
30. Hirokawa, N., Middeldorp, A.: Dependency pairs revisited. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 249–268. Springer, Heidelberg (2004)
31. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. Information and Computation 205(4), 512–534 (2007)
32. Blanqui, F.: CoLoR project (INRIA Lorraine, France) <http://color.loria.fr/>
33. Urbain, X., Forest, J., Courtieu, P.: The A3PAT project (Cédric, CNAM, Paris, France) <http://www3.ensiie.fr/~urbain/a3pat>
34. Podelski, A.: the TERMINATOR project (Microsoft Research, Cambridge, UK) <http://research.microsoft.com/TERMINATOR/>

35. Zantema, H., Waldmann, J.: Termination by quasi-periodic interpretations. In: Baader, F. (ed.) RTA'07. LNCS, Springer, Heidelberg (2007)
36. Contejean, E., Marché, C., Rabehasaina, L.: Rewrite systems for natural, integral, and rational arithmetic. In: Comon, H. (ed.) Rewriting Techniques and Applications. LNCS, vol. 1232, Springer, Heidelberg (1997)
37. Deplagne, É.: Sequent Calculus Viewed Modulo. In: Pilière, C. (ed.) ESSLLI Student Session, Univ. Birmingham, pp. 66–76 (2000)
38. Zantema, H.: Termination of term rewriting by semantic labelling. *Fundamenta Informaticae* 24, 89–105 (1995)
39. Cohen, D., Watson, P.: An efficient representation of arithmetic for term rewriting. In: Book, R.V. (ed.) RTA'91. LNCS, vol. 488, pp. 240–251. Springer, Heidelberg (1991)

Random Descent

Vincent van Oostrom

Universiteit Utrecht, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
oostrom@phil.uu.nl

Abstract. We introduce a method for establishing that a reduction strategy is normalising and minimal, or dually, that it is perpetual and maximal, in the setting of abstract rewriting. While being complete, the method allows to reduce these global properties to the verification of local diagrams. We show its usefulness both by giving uniform proofs of some known results and by establishing new ones.

1 Introduction

Consider the following two natural combinations of properties of strategies.

- normalisation (constructs a reduction to normal form from an object if it exists) and minimality (normal forms are reached in the minimal number of steps).
- perpetuality (constructs an infinite reduction from an object if it exists) and maximality (normal forms are reached in the maximal number of steps).

The former combination is of interest when implementing a rewrite system, whereas the latter is useful for its complexity analysis. Although both have received attention for various concrete strategies and rewrite systems, to our knowledge no general proof method for establishing them has been developed.

In this paper we develop methods for comparing strategies \blacktriangleright and \blacktriangleright in two ways, the former being more appropriate for rewrite systems having unique normal forms, the latter for systems where normal forms need not be unique:

- $(\forall\forall)$ For every pair of maximal \blacktriangleright - and \blacktriangleright -reductions from the same object, the length of the first does not exceed that of the second.
- $(\forall\exists)$ For every maximal \blacktriangleright -reduction from an object, there exists a maximal \blacktriangleright -reduction from that object which is at least as long.

The main contributions of this paper are firstly, the reduction of both combinations of properties of the first paragraph to the $(\forall\forall)$ -comparison problem, and secondly the further reduction of both comparison problems which are global (quantifying over all reductions from an object), to local properties (quantifying only over all steps from an object).

To illustrate the power of our methods, we use examples and results from the literature. Other than that, we assume only basic rewriting knowledge.

2 Abstract Rewrite Systems and Strategies

We recapitulate from [1] Chs. 8 and 9] the main notions from rewriting employed, introduce some new ones, and fix our notations. For background, motivation, and further pointers to the literature, we refer the reader to the mentioned chapters, as both our fundamental notions of ARS and strategy (going back at least to [2,3]) seem to be missing from other textbooks on rewriting.

Definition 1. *An abstract rewrite system (ARS) is a system consisting of a set of objects, a set of steps, and source and target functions mapping steps to objects [1, Def. 8.2.2].*

We employ arrow-like notations to denote ARSs, e.g. \rightarrow , \triangleright , \blacktriangleright , \rightsquigarrow . That ϕ is a step of \rightarrow from a to b , i.e. having a as source and b as target, is denoted by $\phi:a \rightarrow b$ or $a \rightarrow_\phi b$. We may omit the step, the source, or the target from this notation when irrelevant, e.g. $a \rightarrow_\phi$ indicates ϕ is a step from a , and $a \rightarrow b$ that there is a step from a to b . A normal form is an object which is not the source of a step. An object is deterministic if it is the source of at most one step. An ARS is deterministic if all objects are. We say \blacktriangleright is a sub-ARS of \rightarrow , denoted by $\blacktriangleright \subseteq \rightarrow$, if the set of objects/steps of \blacktriangleright is a subset of the set of objects/steps of \rightarrow , and the domain/source map of \blacktriangleright is the restriction of that of \rightarrow .

Remark 1. We follow [2] in employing the intensional notion of abstract rewrite system, instead of the extensional notion of rewrite relation. Whereas the former allows for distinct steps between the same two objects, the latter does not. For instance, in the abstract rewrite system generated by the rule $I(x) \rightarrow x$ there are two distinct steps from $I(I(a))$ to $I(a)$, one corresponding to contracting the outer redex, the other to contracting the inner redex, whereas in the rewrite relation these are (necessarily) confounded. As a consequence, strategies such as the innermost strategy could not be expressed faithfully at the abstract level if we were to employ rewrite relations.

Using the above we present the main notion of this paper, that of a strategy.

Definition 2. *A strategy for an ARS \rightarrow is a sub-ARS of \rightarrow having the same sets of objects and normal forms [1, Def. 9.1.1].*

The idea is that a strategy corresponds to making a choice among the steps possible at each object. The choice may leave all possibilities open (\rightarrow is a strategy for itself), but not decline all (as that would create normal forms).

Example 1. There are exactly three strategies for the ARS $a \leftrightarrow b \rightarrow c$: the ARS itself, $a \rightarrow b \rightarrow c$, and $a \leftrightarrow b \quad c$ [1, Exc. 9.1.3]. Note that e.g. the sub-ARS $a \leftarrow b \rightarrow c$ is not a strategy as it turns a into a normal form.

Remark 2. Our notion of strategy is the intensional version of the extensional notion in [3]. Like there, we do not impose additional conditions such as determinism or computability often found in the literature. Determinism would preclude expressing *the* (as opposed to *an*) innermost strategy. Computability would preclude e.g. the internal needed strategy below from being a strategy.

Henceforth \triangleright and \blacktriangleright are assumed to be strategies for \rightarrow .

As the notions of \triangleright -, \rightarrow -, and \blacktriangleright -normal form are all the same, we simply speak of normal forms. Since strategies are ARSs themselves, all ARS notions and the ARS constructions below apply to them. The converse of an ARS is obtained by swapping the source and target of each step, and denoted by the mirror-image of its notation, e.g. \leftarrow denotes the converse of \rightarrow . The union of two ARSs is obtained by taking unions componentwise, and denoted by the union of the notations, e.g. \leftrightarrow denotes the union of \leftarrow and \rightarrow . A reduction is either finite or infinite. A finite reduction from a to b is inductively defined as being either the empty reduction a and then $a = b$, or a step from a to c followed by a finite reduction from c to b , for some c . An *infinite* reduction from a is coinductively defined as a step from a to b followed by an infinite reduction from b , for some b . The reduction ARS, generated by taking the finite reductions as steps, is denoted by the repetition of the notation of the original ARS, e.g. \rightarrow generates \rightarrow . (If the repetition would lead to clutter, we affix a superscripted $*$ instead, e.g. \leftrightarrow generates \leftrightarrow^* .) Concatenating a finite reduction \mathcal{R} and a reduction \mathcal{S} is defined in the usual way by induction on \mathcal{R} and denoted by $\mathcal{R}\cdot\mathcal{S}$. The length of a reduction, obtained by counting the steps in it, is either finite (a natural number) or infinite (ω). A reduction is maximal if it either is a finite reduction to normal form or infinite. An object is terminating if it only allows finite reductions. An ARS is terminating, if all objects are. We will indicate (constraints on) the length of a reduction by superscripting, e.g. $\mathcal{R}:a \rightarrow^{\leq 5} b$ indicates that \mathcal{R} is a reduction of length at most 5 from a to b , and $\mathcal{S}:a \rightarrow^\omega$ that \mathcal{S} is an infinite reduction from a . A conversion is a finite \leftrightarrow -reduction, and we call the generated \leftrightarrow^* the conversion ARS. The *distance* $d(\mathcal{R})$ of a conversion \mathcal{R} is the number (an integer) of \rightarrow -steps minus the number of \leftarrow -steps in \mathcal{R} . An ARS is said to have unique normal forms if every object has a conversion to at most one normal form.

Remark 3. It would be interesting to extend our results below to reductions of transfinite length (applicable to concrete systems as those in [11, Ch. 12]). That would require developing an intensional version of transfinite ARSs first.

We conclude these preliminaries with formalizing the properties of strategies we would like to establish, as discussed in the first paragraph of the introduction. We already use \triangleright and \blacktriangleright according to the rôle they will play below.

Definition 3. – \triangleright is normalising, if every object from which there is an \rightarrow -reduction to normal form, only allows finite maximal \triangleright -reductions.

- \triangleright is minimal, if the length of any \triangleright -reduction from an object to normal form, is minimal among the \rightarrow -reductions from the former to the latter.
- \blacktriangleright is perpetual, if every object from which there is an infinite \rightarrow -reduction, only allows infinite maximal \blacktriangleright -reductions.
- \blacktriangleright is maximal, if the length of any \blacktriangleright -reduction from an object to normal form, is maximal among the \rightarrow -reductions from the former to the latter.

3 Comparing Strategies Universally

We introduce a method to compare strategies \triangleright and \blacktriangleright for a rewrite system \rightarrow .

Definition 4. \triangleright is universally better than \blacktriangleright , if for every object a , and every pair of maximal \triangleright - and \blacktriangleright -reductions from a , the length of the first does not exceed that of the second.

As we use the adverb ‘universally’ only to distinguish the current notion of better from the one to be introduced in the next section, we will elide it in the rest of the present section. Being better than is transitive, but not an order.

Example 2. Setting both \triangleright and \blacktriangleright to the ARS $a \xrightarrow{\overline{b}} c$ shows a strategy need not be self-better, i.e. better than itself. The reason for failure is that there are reductions of distinct lengths from the object a to its normal form c .

Letting \triangleright and \blacktriangleright be obtained from the ARS $a \rightarrow b_i \rightarrow c$ with $i \in \{1, 2\}$, by omitting either of the steps $a \rightarrow b_i$, shows failure of anti-symmetry: the strategies are distinct but each is better than the other.

Removing the step from a to b in the first part of the example yields a strategy which is both better and self-better (cf. Theorem 3). These exist in general:

Proposition 1. Each ARS has a better strategy which is self-better.

Proof. For an ARS \rightarrow , let WN_i be $\{a \mid i = \mu n.a \rightarrow^n \cdot \dashv\}$, the set of objects whose shortest reduction to normal form has length i . The strategy \triangleright is obtained from \rightarrow by omitting all steps from WN_{1+i} to the complement of WN_i . As by definition each object in WN_{1+i} has some step to WN_i , \triangleright is a strategy for \rightarrow . It is better than both \rightarrow and itself since every maximal \triangleright -reduction from an object in WN_i has length i , and the other objects only allow infinite maximal reductions. \square

When applied to the ARS in the second part of Example 2 the construction yields the ARS itself. More generally, it yields the largest better strategy which is self-better. Note that, dually, a self-better strategy \blacktriangleright for \rightarrow with \rightarrow better than \blacktriangleright , exists, but only for \rightarrow finitely branching (FB). Leaving to future research a more thorough study of the better relation, we proceed by linking it to the two combinations of properties in the first paragraph of the introduction.

Theorem 1. If \triangleright is better than \blacktriangleright , then:

- \triangleright is normalising and minimal, in case $\blacktriangleright = \rightarrow$.
- \blacktriangleright is perpetual and maximal, in case $\rightarrow = \triangleright$.

The reverse implication holds in case \rightarrow has unique normal forms.

Proof. Let \mathcal{R} and \mathcal{S} be maximal \triangleright - and \blacktriangleright -reductions from a .

- ‘Only if’: Suppose \mathcal{S} ends in some normal form b . By the assumption that \triangleright is better than $\blacktriangleright = \rightarrow$, the length of \mathcal{S} is an upper bound on the length of any \triangleright -reduction from a (normalisation), in particular on that of \mathcal{R} (minimality). ‘If’: Since otherwise there is nothing to prove, suppose \mathcal{S} ends in some normal form b . By normalisation of \triangleright , by $\blacktriangleright = \rightarrow$, and maximality of \mathcal{R} , then \mathcal{R} must also end in some normal form, which by uniqueness of normal forms is equal to b , from which we conclude by minimality of \triangleright .

- ‘Only if’: By the assumption that $\rightarrow = \triangleright$ is better than \blacktriangleright , the length of \mathcal{R} is a lower bound on the length of any \blacktriangleright -reduction from a (perpetuality), in particular on that of \mathcal{S} if \mathcal{S} ends in some normal form (maximality).
 ‘If’: Since otherwise there is nothing to prove, suppose \mathcal{S} ends in some normal form b . By perpetuality of \blacktriangleright , by $\rightarrow = \triangleright$, and maximality of \mathcal{R} , then \mathcal{R} must also end in some normal form, which by uniqueness of normal forms is equal to b , from which we conclude by maximality of \blacktriangleright .

Remark 4. That the reverse implication needs uniqueness of normal forms to hold, could indicate that our notion of being better is ‘too universal’; even lengths of reductions to distinct normal forms are compared. It does not seem to make much sense in general to do so (think of an ARS modelling a non-deterministic choice between otherwise unrelated computations). We leave it to future research to investigate relativising comparisons to the result (normal form) computed.

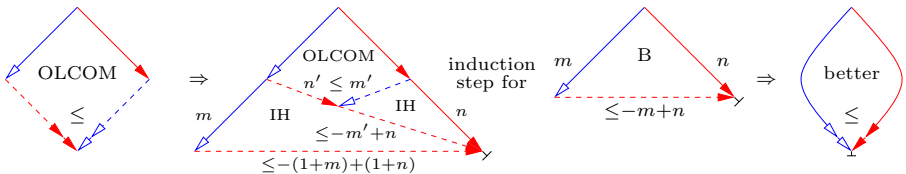
The theorem suggests that for establishing either of the combinations, normalisation and minimality or perpetuality and maximality, a single proof method for establishing that one strategy is better than another might suffice. It does.

Definition 5. *If $a \triangleleft \blacktriangleright b$ implies either $\triangleleft^\omega b$ or $a \blacktriangleright^n \triangleleft^m b$ for some $n \leq m$, then \triangleright ordered locally commutes with \blacktriangleright , abbreviated to $OLCOM(\triangleright, \blacktriangleright)$.*

Our reduction of the global property of ‘being better’ to the local property $OLCOM$ is analogous to the way in which Newman’s Lemma reduces the global property of confluence to the local property of local confluence [2] (more accurately, to the reduction of confluence to local decreasingness [4], as that method is complete). Indeed, $OLCOM$ can be viewed as obtained from local confluence (more precisely, local commutation) by enriching the latter with an ordering constraint on the lengths of the reductions: the length of the ‘left-reduction’ does not exceed the length of the ‘right-reduction’. Note that gluing two such diagrams together in the usual way, yields a diagram again satisfying the ordering constraint. We leave the study of these diagrams to future research. Below we will treat the left disjunct of $OLCOM$, i.e. the case $\triangleleft^\omega b$, as a ‘degenerate’ case.

Theorem 2. *$OLCOM(\triangleright, \blacktriangleright)$ only if \triangleright is better than \blacktriangleright . The reverse holds if \triangleright or \blacktriangleright is equal to \rightarrow and \rightarrow has unique normal forms.*

Proof. ‘Only if’: We successively show the two implications $OLCOM(\triangleright, \blacktriangleright) \Rightarrow B(\triangleright, \blacktriangleright)$, \triangleright is bounded by \blacktriangleright , and $B(\triangleright, \blacktriangleright) \Rightarrow \triangleright$ is better than \blacktriangleright . In a diagram:



Here $B(\triangleright, \blacktriangleright)$ is defined as: for each $b \triangleleft^m a \blacktriangleright^n c$ with c in normal form, $b \blacktriangleright^{\leq -m+n} c$. We show it holds by induction on n , assuming $OLCOM(\triangleright, \blacktriangleright)$. If $m = 0$, it is

trivial. Otherwise, $a \neq c$ as $\blacktriangleright, \blacktriangleleft$ -normal forms coincide, so $b \blacktriangleleft^m b' \blacktriangleleft a \blacktriangleright c' \blacktriangleright^n c$. By $\text{OLCOM}(\blacktriangleright, \blacktriangleright)$ either $\blacktriangleleft^\omega c'$ or $b' \blacktriangleright^{n'} d \blacktriangleleft^{m'} c'$ for some d and $n' \leq m'$. The former cannot hold as it would entail $d' \blacktriangleleft^{n+1} c' \blacktriangleright^n c$ for some d' , so by the induction hypothesis $d' \blacktriangleright^{-1} c$, which cannot be. In case of the latter the induction hypothesis can be applied to $d \blacktriangleleft^{m'} c' \blacktriangleright^n c$, yielding a reduction $d \blacktriangleright^k c$ with $k \leq -m' + n$. Since $n' + k \leq n' + -m' + n \leq n$, the induction hypothesis can be applied to $b \blacktriangleleft^m b' \blacktriangleright^{n'} d \blacktriangleright^k c$, yielding a reduction $b \blacktriangleright \blacktriangleright c$ bounded in length by $-m + n' + k \leq -m + n = -(1 + m) + (1 + n)$.

To show the second implication, let \mathcal{R}, \mathcal{S} be maximal $\blacktriangleright, \blacktriangleright$ -reductions from a . If \mathcal{S} is infinite, there is nothing to prove. Otherwise, it is finite and ends by maximality in some normal form, say it has length n and ends in c . Then n is an upper bound on the length of \mathcal{R} , since the length of a mediating reduction as required by $\text{B}(\blacktriangleright, \blacktriangleright)$ can only be non-negative, and by maximality \mathcal{R} ends in c .

‘If’: Let $b \blacktriangleleft a \blacktriangleright c$ and split cases depending on whether $\rightarrow = \blacktriangleright$ or $\blacktriangleright = \rightarrow$.

If $\rightarrow = \blacktriangleright$, let \mathcal{R} and \mathcal{S} be obtained by extending $a \blacktriangleright b$ and $a \blacktriangleright c$ by maximal \blacktriangleright -reductions. If \mathcal{S} is infinite, then the left disjunct of $\text{OLCOM}(\blacktriangleright, \blacktriangleright)$ holds by viewing the \blacktriangleright -steps in the extension part of \mathcal{S} as \blacktriangleright -steps, using that \blacktriangleright was assumed a strategy for $\rightarrow = \blacktriangleright$. If \mathcal{S} is finite, it ends by maximality in some normal form, say d . Now viewing the \blacktriangleright -steps in the extension part of \mathcal{R} as \blacktriangleright -steps, and using the assumption that \blacktriangleright is better than \blacktriangleright , yields that the length of \mathcal{R} does not exceed that of \mathcal{S} . By maximality, \mathcal{R} must end in a normal form, which must be equal to d by uniqueness of normal forms. Hence the right disjunct of $\text{OLCOM}(\blacktriangleright, \blacktriangleright)$ holds via \mathcal{R}, \mathcal{S} .

If $\rightarrow = \blacktriangleright$, then we proceed dually by extending $a \blacktriangleright b$ and $a \blacktriangleright c$ by maximal \blacktriangleright -reductions, and subsequently viewing \blacktriangleright -steps as \blacktriangleright -steps. □

Combining Theorems [11](#) and [12](#) yields that OLCOM may be used to establish both normalisation and minimality as well as perpetuality and maximality for given strategies for an ARS \rightarrow . Apart from being the first method to reduce these global properties (universally quantifying over all reductions) to a local property (OLCOM universally quantifies only over pairs of steps), the method is even complete (it *is* applicable) in case \rightarrow has unique normal forms.

We now illustrate the power of the method by giving uniform proofs of results for concrete rewrite systems from the literature, and by answering an open problem. The first two examples concern normalisation and minimality, the next two examples concern perpetuality and maximality, and the final one both.

Example 3. The innermost strategy for a TRS allows to contract only those redexes in a term which are innermost among all redexes. For instance, in the TRS with rules $a \rightarrow b$ and $d(x) \rightarrow g(x, x)$, the innermost strategy only allows to contract the a -redex in the term $d(a)$. Intuitively, the innermost strategy is *efficient* since it avoids duplication; the innermost reduction $d(a) \rightarrow d(b) \rightarrow g(b, b)$ is shorter than the non-innermost reduction $d(a) \rightarrow g(a, a) \rightarrow g(b, a) \rightarrow g(b, b)$, since contracting the d -redex first causes duplication of the a -redex. The catch is that the innermost strategy also avoids erasure; in the TRS with rules $a \rightarrow f(a)$ and $f(x) \rightarrow b$, an innermost reduction from $f(a)$ only contracts a -redexes and never reach the normal form b , whereas the latter could be reached

efficiently, even in a single f -step, by erasing the a -redex. Combining these two ideas results in a strategy due to Khasidashvili, here denoted by \triangleright and given by:

(internal needed strategy) contract an innermost redex among *needed* ones.

A redex is needed, in Huet and Lévy's sense, if it must be reduced in any reduction to normal form; cf. [11, Sect. 9.4.7]. Note erasable redexes are not needed.

How to show that \triangleright is a normalising and minimal strategy for ordinary rewriting \rightarrow , in case of an orthogonal TRS? We must first show that \triangleright is indeed a strategy for \rightarrow . This follows from the fact (due to Huet and Lévy) that in an orthogonal TRS, any term not in normal form contains a needed redex, hence also an innermost such (but since neededness is not computable, neither is \triangleright).

To show normalisation and minimality of \triangleright , it suffices by the theorem to show $\text{OLCOM}(\triangleright, \rightarrow)$. This we show by a critical pair analysis, distinguishing cases on the relative positions p, q of the redexes contracted in $s \triangleleft_p t \rightarrow_q u$:

- (=) Then the steps are the same by orthogonality, so $s = u$ and we conclude.
- (||) Then $s \rightarrow_q v \triangleleft_p u$, for some term v . Since at least one of the residuals of a needed redex is needed if it is not contracted itself, and by $p \parallel q$ the needed redex at p in t has a unique residual at p in u , that unique residual must therefore be needed. Since the latter is also innermost among the needed redexes in u (being needed or not was not changed for redexes below p), it holds $s \rightarrow_q v \triangleleft_p u$, from which we conclude.
- (<) Then q is non-needed since p was assumed to be the position of a redex innermost among the needed ones. Consider a maximal \triangleright -reduction \mathcal{R} extending $t \triangleright_p s$. Consider the projection \mathcal{S} of \mathcal{R} over the non-needed step $t \rightarrow_q u$. Then \mathcal{S} is a \triangleright -reduction from u of exactly the same length as \mathcal{R} . This follows from the general theory of neededness, in particular from the fact that contracting a non-needed redex can neither erase, nor duplicate, nor create needed redexes. If \mathcal{R} is infinite, then \mathcal{S} is infinite as well so the left disjunct of $\text{OLCOM}(\triangleright, \rightarrow)$ holds. Otherwise, \mathcal{R} ends in a normal form by maximality, hence \mathcal{S} being its projection ends in the same normal form, and the right disjunct of $\text{OLCOM}(\triangleright, \rightarrow)$ holds.
- (>) Then $s \rightarrow_q v \triangleleft_p u$ for some v , obtained by projecting the steps over one another. Per construction of the projection for orthogonal rewrite systems (essentially going back to Church and Rosser), $v \triangleleft_p u$ is in fact a reduction contracting the set of residuals of the redex at position p , which is a set of disjoint positions in the case of TRSs, that is $v \triangleleft_p u$. Partitioning P into sets of non-needed and needed residuals yields a decomposition of $v \triangleleft_p u$ as $v \triangleleft_p v' \triangleleft_p u$, with the \triangleright -reduction being non-empty since p has at least one needed residual in u as it was not contracted in $t \rightarrow_q u$. Finally, consider a maximal \triangleright -reduction \mathcal{R} from v' . If \mathcal{R} is infinite, the left disjunct of $\text{OLCOM}(\triangleright, \rightarrow)$ holds. Otherwise \mathcal{R} ends in a normal form, and, as before, projecting \mathcal{R} over $v \triangleleft_p v'$ yields a reduction from v of exactly the same length and ending in the same normal form, from which we conclude. \square

Since an orthogonal TRS is confluent, it has unique normal forms. Hence *that* normalisation and minimality of the internal needed strategy are reducible to

OLCOM($\triangleright, \rightarrow$) is clear by completeness of our method. The point of the examples is rather to show that a clear methodology for proving OLCOM suggests itself: the case-analysis required for OLCOM often resembles a *critical pair analysis*.

Remark 5. It would be interesting to have a critical pair *lemma* for strategies corresponding to OLCOM in the same way Huet’s Critical Pair Lemma for TRSs corresponds to local confluence. That requires a suitable strategy formalism.

Example 4. In multi-step rewriting, denoted by \twoheadrightarrow , an arbitrary (non-zero) number of redexes in a term may be contracted at the same time. For instance, in the TRS with rules $a \rightarrow b$ and $d(x) \rightarrow g(x, x)$ as above, we have $d(d(a)) \twoheadrightarrow d(g(b, b))$ by contracting the inner d -redex and the a -redex at the same time. For the notion of ‘contracted at the same time’ to make sense, the redexes need to be consistent to each other. (E.g. what would it mean to contract both redexes at the same time in the term $f(g(a))$ in the TRS with rules $f(g(x)) \rightarrow b$ and $g(a) \rightarrow c$?) For orthogonal TRSs this is guaranteed giving a greedy strategy \triangleright :

(full-substitution strategy) contract *all* redexes in the term simultaneously.

How to show that being greedy is best, i.e. that \triangleright is a normalising and minimal strategy for multi-step rewriting \twoheadrightarrow in case of an orthogonal TRS? That \triangleright is a strategy follows from orthogonality, since it guarantees that if *some* multi-step exists, contracting *all* redexes makes sense/is possible, cf. [1, Def. 4.9.5(v)].

To show normalisation and minimality of \triangleright , it suffices by the theorem to show OLCOM($\triangleright, \twoheadrightarrow$). If $s \triangleleft t \twoheadrightarrow u$, then the set of redexes contracted in the multi-step $t \twoheadrightarrow u$ is contained in the set of all redexes in t , by orthogonality. If the sets are the same, then $s = u$ and we are done. Otherwise, by standard residual theory, $s \leftarrow_P u$ where P is the set of residuals after $t \twoheadrightarrow u$ of the set of all redexes in t . If P is the set of all redexes in u , then in fact $s \triangleleft u$ and we are done again. Otherwise, we conclude from $s \twoheadrightarrow v \triangleleft u$, where $s \twoheadrightarrow v$ contracts the set of residuals after $s \leftarrow_P u$ of the set of all redexes in u . \square

The proof in Example 4 goes through for the λ -calculus (the full-substitution strategy is known there as Gross–Knuth reduction) and more generally to orthogonal higher-order pattern rewrite systems, as it only depends on a modicum of residual theory (e.g. [1, Thm. 11.6.29] in the case of higher-order rewriting).

Example 5. The outermost strategy for a rewrite system allows to contract only those redexes in a term which are outermost among all redexes. Intuitively, the outermost strategy is *inefficient* since it promotes duplication; the outermost-reduction $d(a) \rightarrow g(a, a) \rightarrow g(b, a) \rightarrow g(b, b)$ is longer than the non-outermost reduction $d(a) \rightarrow d(b) \rightarrow g(b, b)$ in the TRS with rules $a \rightarrow b$ and $d(x) \rightarrow g(x, x)$. One catch is that the outermost strategy also promotes erasure; in the TRS with rules $a \rightarrow f(a)$ and $f(x) \rightarrow b$, an outermost reduction from $f(a)$ immediately reaches the normal form b , whereas it would be infinitely more *inefficient* to repeat contracting a -redexes. Another catch is that outermost redexes may turn into non-outermost redexes; although the rightmost a -redex in $f(a, a)$ is outermost for the TRS with rules $a \rightarrow b$, $f(b, x) \rightarrow g(x, x)$, its contraction should

be delayed if one strives for *inefficiency*, contracting instead the leftmost a -redex (turning the rightmost a -redex into an innermost one) first and next the f -redex. Combining these three ideas results in a strategy \blacktriangleright due to Khasidashvili:

(limit strategy) contract an *external* redex which does not erase a reducible argument, otherwise recur on such an argument; cf. [11, Sect. 9.5.1].

Externality is Huet and Lévy's notion of a redex which remains outermost until contracted. In the term $f(a)$ above, the f -redex is external, but since contracting it would erase its, reducible, argument a , the limit strategy recurs on it.

That \blacktriangleright is a perpetual and maximal strategy for \rightarrow was originally proven by Khasidashvili for \rightarrow being (generated by) an orthogonal Expression Reduction System. Here, we give a proof via $\text{OLCOM}(\rightarrow, \blacktriangleright)$ for the closely related formalism of orthogonal fully-extended second-order pattern rewrite systems, i.e. the restriction of Nipkow's higher-order rewrite systems to rules with free variables of second-order. That \blacktriangleright is indeed a strategy for \rightarrow holds since a term not in normal form contains an outermost redex, and (generalising Huet and Lévy's result for TRSs) at least one among the outermost redexes is external.

To show $\text{OLCOM}(\rightarrow, \blacktriangleright)$, we perform a critical pair analysis, distinguishing cases on the relative positions p, q of the redexes contracted in $s \leftarrow_p t \blacktriangleright_q u$.

- (=) Then $s = u$ since at most one left-hand side matches a term. We conclude.
- (||) Then $s \rightarrow_q v \leftarrow_p u$, for some term v . The step $s \rightarrow_q v$ could only fail to be a \blacktriangleright -step, if contraction of q in t was due to a recursive call for some o above it, which is blocked in s . This cannot be, as the residual of o in s still erases the argument q is in. Hence $s \blacktriangleright_q v \leftarrow_p u$.
- (<) Then by definition of \blacktriangleright , p is a redex erasing q and $s \leftarrow_p u$.
- (>) Then by the diamond property for orthogonal multi-steps in PRSs [11, Thm. 11.6.29], $s \rightarrow_q v \leftarrow_P u$ for some v , where P is the set of residuals of p after $t \blacktriangleright_q u$, which is non-empty since q is non-erasing. If $s \rightarrow_q v$ is non-erasing, then in fact $s \blacktriangleright_q v \leftarrow_P u$ and we conclude since the non-empty multi-step $v \leftarrow_P u$ develops into a non-empty reduction $v \leftarrow u$ by the Finite Developments Theorem [11, Thm. 11.5.11]. Otherwise, consider a maximal \blacktriangleright -reduction \mathcal{R} from s obtained by first reducing each argument erased by q in turn to its normal form. If this is finite, the redex at position q has become a \blacktriangleright -redex and we adjoin it to \mathcal{R} , and let v be the resulting term. Now consider performing for each descendant along $t \blacktriangleright_q u$ of each such an erased argument, the same steps as in \mathcal{R} , and do this according to the inside-order of these descendants in u . This guarantees that the resulting reduction has at least the same length as \mathcal{R} . (In the 3rd order case that may fail, see the following remark.) Thus if \mathcal{R} is infinite this yields an infinite reduction from u as well, and we are done. Otherwise, this yields a reduction ending in v by [11, Thm. 11.6.29] and we conclude again. \square

The limit strategy need not be maximal nor perpetual for third-order systems:

¹ An example of this in λ -calculus is $(\lambda x.y)N \leftarrow (\lambda x.(\lambda z.y)x)N \blacktriangleright (\lambda z.y)N$; we first *should* \blacktriangleright -reduce N to normal form, say N' , before to proceed with $(\lambda x,y)N' \blacktriangleright y$.

Remark 6. Consider $t = f(G.g(x.G(x)))$ in the orthogonal fully-extended third-order rewrite system:

$$a \rightarrow b \quad g(x.G(x)) \rightarrow G(a) \quad f(G.H(G)) \rightarrow H(y.c)$$

Starting with contracting the head-redex-pattern yields $t \blacktriangleright g(x.c) \blacktriangleright c$. Observe how contracting the head-redex makes the inside g -redex erasing (something impossible in second-order systems) and that contracting the latter first yields a longer reduction: $t \rightarrow f(G.G(a)) \rightarrow f(G.G(b)) \rightarrow c$.

Example 6. Leftmost-outermost redexes are external in $\lambda\beta$ -calculus. Therefore

(F_∞) contract a leftmost-outermost redex which does not erase a reducible argument, otherwise recur on such an argument.

is a limit strategy, hence maximal and perpetual by the above; cf. [5, Thm. 11]. As on λI -terms the leftmost-outermost strategy is F_∞ , it is maximal and perpetual; cf. [6, Proposition 3.17]. What about F_∞ for λ -calculi with explicit substitutions?

The λx^- -calculus of [7] is a $\lambda\beta$ -calculus with explicit substitution operator $\langle := \rangle$, and rules (of which only the third, not the first, is erasing $(N)!$):

$$\begin{aligned} (\lambda x.M)N &\rightarrow M\langle x:=N \rangle \\ x\langle x:=N \rangle &\rightarrow N & (\lambda y.M)\langle x:=N \rangle &\rightarrow \lambda y.M\langle x:=N \rangle \\ y\langle x:=N \rangle &\rightarrow y & (M_1M_2)\langle x:=N \rangle &\rightarrow M_1\langle x:=N \rangle M_2\langle x:=N \rangle \end{aligned}$$

As any reducible term has a leftmost-outermost redex, the ARS \blacktriangleright induced by F_∞ is a strategy for the ARS \rightarrow induced by λx^- . Since λx^- is a second-order rewriting system [8, Def. 13], to show $\text{OLCOM}(\rightarrow, \blacktriangleright)$ it suffices to adapt the analysis of Example 5. The only property of externality we employed there was that it is preserved for residuals (if any). As that also holds for leftmost-outermostness in λx^- , it suffices to supplement the case-analysis with the (unique) critical pair:

$$C[M\langle x:=N \rangle\langle y:=P \rangle] \leftarrow_p C[(\lambda x.M)N\langle y:=P \rangle] \blacktriangleright_q C[(\lambda x.M)\langle y:=P \rangle N\langle y:=P \rangle]$$

We reason as for $(>)$ in Example 5. In particular, we simulate any \blacktriangleright -reduction \mathcal{R} from the term s on the left by a reduction \mathcal{S} from the term u on the right, which is at least as long. To that end, we first reduce u one step further to $u' = C[M\langle y:=P \rangle\langle x:=N\langle y:=P \rangle \rangle]$. After that, simulation of \mathcal{R} by \mathcal{S} is redex-wise: By definition of F_∞ , a $\langle x:=N \rangle\langle y:=P \rangle$ -closure is only ever distributed (over λ or $@$) in its entirety in \mathcal{R} , which can be simulated in \mathcal{S} by distributing the $\langle y:=P \rangle\langle x:=N\langle y:=P \rangle \rangle$ -closure. In case a $\langle x:=N \rangle\langle y:=P \rangle$ -closure is being applied to a variable in \mathcal{R} , its simulation in \mathcal{S} is defined by cases on the variable:

- (x) Then $x\langle x:=N \rangle\langle y:=P \rangle \blacktriangleright N\langle y:=P \rangle \leftarrow^2 x\langle y:=P \rangle\langle x:=N\langle y:=P \rangle \rangle$.
- (y) Then $y\langle x:=N \rangle\langle y:=P \rangle \blacktriangleright^i y\langle x:=N' \rangle\langle y:=P \rangle \blacktriangleright y\langle y:=P \rangle \blacktriangleright P \blacktriangleright^j P'$ where N', P' are the normal forms of N, P (if any). This can be simulated by $y\langle y:=P \rangle\langle x:=N\langle y:=P \rangle \rangle \rightarrow^i y\langle y:=P \rangle\langle x:=N' \rangle\langle y:=P \rangle \rightarrow P\langle x:=N' \rangle\langle y:=P \rangle \rightarrow^j P'\langle x:=N' \rangle\langle y:=P \rangle \rightarrow^+ P'$, using for the final reduction that neither the variable x nor any closures occur in the normal form P' .
- (z) Analogous to the previous case ending in z (if at all) instead of P' . \square

This solves the open problem [9, Rem. 3.18]. The method can easily be adapted to show maximality and perpetuality of F_∞ for the $\lambda\beta\eta$ -calculus [5, Thm. 19] [2].

Remark 7. Termination proofs for typed λ -calculi (with explicit substitutions) usually involve a ‘syntactic’ commutation property. The strategy F_∞ allows to factor this property out, giving rise to fully ‘semantic’ termination proofs.

We say \rightarrow is *ordered weak Church–Rosser* (OWCR) if $\text{OLCOM}(\rightarrow, \rightarrow)$. OWCR entails \rightarrow is self-better (Theorem [2]) and if an object can be reduced to normal form in n steps, then *any* strategy will do so (Theorem [1]); in Newman’s terminology [2, p. 226]: the end-form is reached by random descent. Inspired by this, we say \rightarrow has *random descent* (RD), if for each $\mathcal{R}:a \leftrightarrow^* b$ with b in normal form, all maximal reductions from a have length $d(\mathcal{R})$ and end in b . Note that for an ARS satisfying OWCR it suffices to prove normalisation to establish both termination and (random descent) confluence; see [10] for an example of this.

Theorem 3 (Random Descent). $\text{OWCR} \Leftrightarrow \text{RD}$.

Proof. ‘Only if’: Setting $\triangleright = \rightarrow = \blacktriangleright$, we obtain $\text{OWCR} \Rightarrow \text{B}(\rightarrow, \rightarrow)$ by using the implication $\text{OLCOM}(\triangleright, \blacktriangleright) \Rightarrow \text{B}(\triangleright, \blacktriangleright)$ in the proof of Theorem [2]. Next, we claim $\text{B}(\rightarrow, \rightarrow)$ implies *self-boundedness* (SB), where SB states that for any conversion $\mathcal{R}:a \leftrightarrow^* b$ with b in normal form, there exists $a \rightarrow^{\leq d(\mathcal{R})} b$. The claim follows by an easy induction on the number of peaks in the conversion (analogous to the way the Church–Rosser property is proven from confluence).

Having established SB, we prove RD. Let $\mathcal{R}:a \leftrightarrow^* b$ with b in normal form, $n = d(\mathcal{R})$ and consider a maximal reduction \mathcal{S} from a . On the one hand, n is an *upper* bound on the length of \mathcal{S} , since otherwise there would be a reduction $\mathcal{S}':a \rightarrow^{n+1} a'$ hence a conversion to normal form $\mathcal{S}'^{-1} \cdot \mathcal{R}:a' \leftrightarrow^* b$ with negative distance -1 , contradicting SB. Thus by maximality \mathcal{S} ends in a normal form b' which by SB for $\mathcal{S}^{-1} \cdot \mathcal{R}:b' \leftrightarrow^* b$ reduces to b , so $b' = b$. On the other hand, n is a *lower* bound on the length of \mathcal{S} , as else $\mathcal{R}^{-1} \cdot \mathcal{S}:b \leftrightarrow^* a$ would be a conversion to normal form having negative distance, contradicting SB.

‘If’: Let $\mathcal{R}:b \leftarrow a \rightarrow c$, and let \mathcal{S} be a maximal reduction from c . If \mathcal{S} is infinite, we are done. Otherwise, it is finite and ends in a normal form, and we conclude from RD using $d(\mathcal{S}) = d(\mathcal{R} \cdot \mathcal{S})$. \square

Instances of calculi for which random descent has been established and its consequences used, abound in the literature, almost invariably proven in ad hoc fashion. Some examples are linear λ -calculi [11, Cor. 3.4], [5, Prop. 33], spine strategies for λ -calculus [6, Prop. 4.21], in/external strategies for orthogonal TRSs [1], orthogonal string rewrite systems [12], orthogonal graph rewrite systems in particular Lafont’s interaction nets [13], and orthogonal process calculi in particular those modelled by Stark’s concurrent transition systems [14].

Several local conditions sufficient for RD, generalizing [2, Thm. 2], have been proposed in the literature, e.g. *balanced weak Church–Rosser* (BWCR [3,15]) *balanced SCR* [16], *linear biclosed* [8], and $\text{SCR}^{\geq 1}$ [16]. However, none is complete

² Only failure of preservation of leftmost-outermostness ($\lambda x.(\lambda y.Kyx)x$) requires care.

as they all fail to cover the ARS $a \rightarrow b \rightarrow c \rightarrow d \rightarrow c \leftarrow a$, which *does* satisfy OWCR. In fact, no global property had been explicitly identified before, but it is the global property upon which all applications are seen to rely. For instance, all results in [3,15] trivially generalise by replacing everywhere BWCR by RD.

Example 7. Let the ARS \rightarrow have finite lists as objects and swapping of adjacent elements in lists-which-are-not-sorted as steps. The strategy \triangleright , which only swaps elements which are out-of-order, so-called *inversions*, has random descent as follows from OWCR(\triangleright), the only interesting case being the critical pair schematically given by $bca \triangleleft cba \triangleright cab$, which is completed as $bca \triangleright bac \triangleright abc \triangleleft acb \triangleleft cab$. Since insertion sort is an instance of \triangleright which is $\Theta(n^2)$, and OLCOM($\triangleright, \rightarrow$) follows by an easy critical pair analysis, we conclude sorting-by-swapping is $\Omega(n^2)$.

4 Comparing Strategies Existentially

We introduce our second (and third) way to compare strategies.

Henceforth it is assumed that $\rightarrow = \blacktriangleright \cup \triangleright$.

Definition 6. *An ordered pair is a pair of maximal reductions from an object, such that if the second ends, the first ends in the same object and is not longer. \mathcal{R} can be completed on the left (right) by \mathcal{S} if \mathcal{S}, \mathcal{R} (\mathcal{R}, \mathcal{S}) is an ordered pair.*

- \triangleright is existentially better than \blacktriangleright , if every maximal \blacktriangleright -reduction can be completed on the left by a \triangleright -reduction.
- \blacktriangleright is existentially worse than \triangleright , if every maximal \triangleright -reduction can be completed on the right by a \blacktriangleright -reduction.

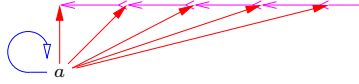
We drop the adverb ‘existentially’ if no confusion with the notion of better from the previous section can arise. Clearly, both better and worse are quasi-orders, but neither needs to be anti-symmetric (for the same reason as before).

Remark 8. For ARSs having unique normal forms, \triangleright being universally better than \blacktriangleright implies both \triangleright being existentially better than \blacktriangleright and \blacktriangleright being existentially worse than \triangleright but not vice versa. Since our existential ways to compare strategies are relativised with respect to the normal forms, whereas the universal way was not, they are in general incomparable for ARSs not having unique normal forms.

The next goal is to reduce the global properties better and worse to local ones.

Definition 7. *- \rightarrow has left extraction (LE) if every finite maximal reduction of shape $\blacktriangleright \cdot \blacktriangleright$ can be completed on the left by a reduction of shape $\triangleright \cdot \rightarrow$.*
- \rightarrow has right extraction (RE) if every finite maximal reduction of shape $\triangleright \cdot \blacktriangleright$ can be completed on the right by a reduction of shape $\blacktriangleright \cdot \rightarrow$.

The idea for LE is that to show \triangleright is better than \rightarrow , it suffices that any *initial* \blacktriangleright -step can be ‘improved’ into a \triangleright -step. The dual idea for RE will need extra assumptions to work, i.e. to imply that \blacktriangleright is worse than \rightarrow , as shown by:



RE holds but $a \triangleright^\omega$ cannot be completed on the right by \blacktriangleright ($\blacktriangleright = \triangleright \cap \blacktriangleright$)

Remark 9. LE, OLCOM, RE are essentially the same diagram, the main difference residing in which sides are existentially quantified (left, bottom, right).

Lemma 1. \rightarrow -non-termination implies \blacktriangleright -non-termination, under the assumptions that $a \triangleright \cdot \triangleright^n$ implies $a \triangleright \cdot \rightarrow^n$ and that \blacktriangleright is finitely branching (FB)

Proof. By FB and König’s Lemma, it suffices to show that for any \rightarrow -reduction there exists a \blacktriangleright -reduction from the same object and of the same length. The proof is by induction on the length of the \rightarrow -reduction. The base case being trivial, suppose it is of shape $a \rightarrow a' \rightarrow^n$. By the induction hypothesis for $a' \rightarrow^n$, there exists a reduction $a' \blacktriangleright^n$. If in fact $a \blacktriangleright a'$, we are done. Otherwise, $a \triangleright a'$ and the assumption for $a \triangleright a' \blacktriangleright^n$ yields $a \blacktriangleright a'' \rightarrow^n$ from which we conclude by the induction hypothesis applied to $a'' \rightarrow^n$. \square

Two sufficient conditions for the lemma are obtained by requiring on top of FB, either RE with ‘finite’ removed from its definition, or [17, Lemma 7].

Theorem 4. If LE, then \triangleright is better than \rightarrow . If RE, then \blacktriangleright is worse than \rightarrow , under the assumptions of Lemma 1.

Proof. To prove the first item, let \mathcal{R} be a maximal reduction from a . If \mathcal{R} is infinite, then we are done. Otherwise \mathcal{R} is finite and we proceed by induction on its length. The base case being trivial, suppose \mathcal{R} is of shape $a \rightarrow a' \rightarrow b$. By the induction hypothesis for $a' \rightarrow b$, we get $a' \blacktriangleright b$ which is no longer, so we are done if in fact $a \triangleright a'$. Otherwise LE for $a \blacktriangleright a' \blacktriangleright b$ yields $a \triangleright a'' \rightarrow b$ which is no longer, so by the induction hypothesis for $a'' \rightarrow b$, $a'' \blacktriangleright b$ which is no longer.

To prove the second item, note that if \rightarrow is non-terminating for a , then Lemma 1 yields an infinite \blacktriangleright -reduction from a and we are done. Otherwise, we proceed as in the first item, but by well-founded induction ordered by \rightarrow on the source a of \mathcal{R} , exchanging longer with shorter, LE with RE, and \triangleright with \blacktriangleright . \square

As a typical application of the above, we present a λ -calculus λ^+ with non-deterministic choice embodied by the rule $M_1 + M_2 \rightarrow M_i$ (cf. [18]).

Lemma 2. RE without ‘finite’ in its condition hold for \blacktriangleright the F_∞ -strategy and \triangleright the reduction relation of λ^+ , with F_∞ as for λ -calculus (Example 6) additionally choosing for $M_1 + M_2$ either to recur on M_i if it is reducible, or to select M_i in case the other argument is in normal form.

Proof. Let $t \triangleright_p s \triangleright^\alpha$ be a maximal reduction and distinguish cases on whether s is a normal form or not. If s is a normal form, we show $t \blacktriangleright_q u \rightarrow s$, for some u . If in fact $t \blacktriangleright_p s$ this is trivial. Otherwise, the step $t \triangleright_p s$ must be outermost and all other redex-patterns in t must be below p and erased by the step. Then for any $t \blacktriangleright t'$ it holds $t' \triangleright_p s$. If s is not a normal form, the reduction is of shape $t \triangleright_p s \blacktriangleright_q u \triangleright^{\alpha-1}$ for some u . It suffices to show that this entails $t \blacktriangleright \cdot \rightarrow^+ u$. Let $t = C[P]_p$ and distinguish cases on the relative positions of p, q .

- (\leq) If $P = (\lambda x.M)N$ then either P is in fact a \blacktriangleright -redex, or x does not occur in M and N is not normal, thus N is erased and we proceed as above. If $P = M_1 + M_2$ and, say, M_1 is selected and then \blacktriangleright -rewritten to M'_1 , then $t \blacktriangleright C[M'_1 + M_2]_p \rightarrow u$.
- ($\not\leq$) If $t \blacktriangleright_q$ then $t \blacktriangleright_q \cdot \rightarrow u$ by [11, Thm. 11.6.22], where in fact the \rightarrow -reduction cannot be empty since \blacktriangleright -steps cannot erase \triangleright -steps. Otherwise, contracting P turns the ‘ q -branch’ of the greatest common predecessor o of p, q in the term-tree, into an ‘ F_∞ -branch’. This can only happen if either the step was in fact a step to normal form of the ‘ p -branch’, or if it created the body of a β -redex at position o . In either case it is a \blacktriangleright -step unless it erases a non-normal argument N , but then we may proceed as before. \square

As any term contains only finitely many occurrences of $+$, \triangleright is finitely branching, and by $\blacktriangleright \subseteq \triangleright$, \blacktriangleright is so too. Thus by Theorem 4 \blacktriangleright is worse than \triangleright . This can be useful to prove termination of typed subcalculi of λ^+ via termination of F_∞ ; cf. Remark 7. We conclude with a case-study of abstract copying.

Definition 8. If $\blacktriangleright = \bigcup_p \blacktriangleright_p$, then \blacktriangleright copies \triangleright if for $b \triangleleft a \blacktriangleright_p$, either $b \triangleleft \cdot \blacktriangleleft_p a$ or q exists with $(a \blacktriangleright_p a'$ implies $b \blacktriangleright_q \cdot \triangleleft^+ a'$, and $b \blacktriangleright_q b'$ implies $a \blacktriangleright_p \cdot \triangleright^+ b')$.

E.g. the outermost strategy copies $\rightarrow_{\mathcal{T}}$ for $\mathcal{T} = \{f(x) \rightarrow g(x), f(x) \rightarrow h(x, x)\}$.

Theorem 5. If \blacktriangleright copies \triangleright , then \blacktriangleright is worse than \triangleright .

Proof. Let $a_0 \blacktriangleright_p$ and let $a_0 \triangleright a_1 \triangleleft^\alpha$ be maximal. It suffices to find $a_0 \blacktriangleright b \triangleright^\beta$ which is no shorter and ends in the same normal form if at all, as repeating this on b leads to an ever growing \blacktriangleright -reduction from a_0 , while preserving the property.

Setting $p_0 = p$, construct a maximal sequence of indices p_i such that $(a_i \blacktriangleright_{p_i} a'$ implies $a_{i+1} \blacktriangleright_{p_{i+1}} \cdot \triangleleft^+ a'$, and $a_{i+1} \blacktriangleright_{p_{i+1}} b'$ implies $a_i \blacktriangleright_{p_i} \cdot \triangleright^+ b')$. Look for the first i for which p_{i+1} is not defined.

If it exists, then a_i cannot be in normal form since $a_i \blacktriangleright_{p_i}$ holds by induction on i with base case $a_0 \blacktriangleright_{p_0}$. Hence by maximality $a_i \triangleright a_{i+1}$ implying $a_i \blacktriangleright_{p_i} b_i \triangleright a_{i+1}$ for some b_i . Per construction, there exists b_{i-1} such that $a_{i-1} \blacktriangleright_{p_{i-1}} b_{i-1} \triangleright^+ b_i$. Continuing in this fashion by induction on i , yields $a_0 \blacktriangleright_{p_0} b_0 (\triangleright^+)^i b_i \triangleright a_{i+1} \triangleright^{\alpha+i}$ which is as desired.

If it doesn't exist, then select an arbitrary step $a_0 \blacktriangleright_{p_0} b_0$. Per construction, there exists b_1 such that $a_1 \blacktriangleright_{p_1} b_1 \triangleleft^+ b_0$. Continuing in this fashion by induction, yields $a_0 \blacktriangleright_{p_0} b_0 \triangleright^+ b_1 \triangleright^+ b_2 \triangleright^\omega$ which has the desired property. \square

It is easy to see that for TRSs the innermost strategy not only copies itself, but also copies, hence by Theorem 5 is worse than, non-dup-generalized innermost rewriting \rightarrow_{ndg} [17], generalizing all results on \rightarrow_{ndg} in [17] to the non-finitely branching case. For another application of Theorem 5: any positional [11, p. 512] innermost strategy copies the innermost strategy, hence termination of the former implies termination of the latter, simplifying [19, Thm. 6] and [20, Thm. 2].

Acknowledgments. I thank all people who have supplied constructive criticism to this paper since its initial conception in 2004. In particular, I thank F.J. de Vries for asking the initial question, J. Ketema for his careful reading of a previous version, and A. Visser for asking the final question.

References

1. Terese.: Term Rewriting Systems. Cambridge University Press, Cambridge (2003)
2. Newman, M.: On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43(2), 223–243 (1942)
3. Toyama, Y.: Strong sequentiality of left-linear overlapping term rewriting systems. In: *Proceedings of the 7th LICS*, pp. 274–284. IEEE Computer Society Press, Los Alamitos (1992)
4. Oostrom, V.v.: Confluence by decreasing diagrams. *Theoretical Computer Science* 126(2), 259–280 (1994)
5. Sørensen, M.: Efficient longest and infinite reduction paths in untyped λ -calculi. In: Kirchner, H. (ed.) *CAAP 1996*. LNCS, vol. 1059, pp. 287–301. Springer, Heidelberg (1996)
6. Barendregt, H., Kennaway, R., Klop, J., Sleep, M.: Needed reduction and spine strategies for the lambda calculus. *Information and Computation* 75(3), 191–231 (1987)
7. Bloo, R.: Preservation of Termination for Explicit Substitution. PhD thesis, Technische Universiteit Eindhoven (1997)
8. Khasidashvili, Z., Ogawa, M., van Oostrom, V.: Uniform Normalisation beyond Orthogonality. In: Middeldorp, A. (ed.) *RTA 2001*. LNCS, vol. 2051, pp. 122–136. Springer, Heidelberg (2001)
9. Bonelli, E.: Substitutions explicites et réécriture de termes. PhD thesis, Paris XI (2001)
10. Oostrom, V.v.: Bowls and Beans, Available from author's homepage (2004)
11. Simpson, A.: Reduction in a linear lambda-calculus with applications to operational semantics. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 219–234. Springer, Heidelberg (2005)
12. Jantzen, M.: Confluent String Rewriting. *EATCS Monographs on Theoretical Computer Science*, vol. 14. Springer, Heidelberg (1988)
13. Lafont, Y.: Interaction nets. In: *Proceedings of the 17th POPL*, pp. 95–108. ACM Press, New York (1990)
14. Stark, E.: Concurrent transition systems. *Theoretical Computer Science* 64, 221–269 (1989)
15. Toyama, Y.: Reduction strategies for left-linear term rewriting systems. In: Middeldorp, A., van Oostrom, V., van Raamsdonk, F., de Vrijer, R. (eds.) *Processes, Terms and Cycles: Steps on the Road to Infinity*. LNCS, vol. 3838, pp. 198–223. Springer, Heidelberg (2005)
16. Gramlich, B.: On some abstract termination criteria, WST '99 Talk (1999)
17. Pol, J.v.d., Zantema, H.: Generalized innermost rewriting. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 2–16. Springer, Heidelberg (2005)
18. de'Liguoro, U., Piperno, A.: Nondeterministic extensions of untyped λ -calculus. *Information and Computation* 122(2), 149–177 (1995)
19. Krishna Rao, M.: Some characteristics of strong innermost normalization. *Theoretical Computer Science* 239, 141–164 (2000)
20. Fernández, M.L., Godoy, G., Rubio, A.: Orderings for innermost termination. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 17–31. Springer, Heidelberg (2005)

Correctness of Copy in Calculi with Letrec

Manfred Schmidt-Schauß

FB Informatik und Mathematik, Institut für Informatik, J.W.Goethe-Universität,
Robert-Mayer-Str 11-15, Postfach 11 19 32,
D-60054 Frankfurt, Germany
`schauss@ki.informatik.uni-frankfurt.de`

Abstract. Call-by-need lambda calculi with letrec provide a rewriting-based operational semantics for (lazy) call-by-name functional languages. These calculi model the sharing behavior during evaluation more closely than let-based calculi that use a fixpoint combinator. However, currently the knowledge about correctness w.r.t. observational equivalence of modifying the sharing in letrec-based calculi is full of gaps. In this paper we develop a new proof method based on a calculus on infinite trees, generalizing the parallel 1-reduction, for showing correctness of instantiation operations. We demonstrate the method in the small calculus $LR\lambda$ and show that copying at compile-time can be done without restrictions. We also show that the call-by-need and call-by-name strategies are equivalent w.r.t. contextual equivalence. A consequence is correctness of all the transformations like instantiation, inlining, specialization and common subexpression elimination in $LR\lambda$. The result for $LR\lambda$ also gives an answer to unresolved problems in several papers and thus contributes to the knowledge about deterministic calculi with letrec.

The method also works for a calculus with case and constructors, and also with parallel or. We are also confident that the method scales up for proving correctness of copy-related transformations in non-deterministic lambda calculi if restricted to “deterministic” subterms.

1 Introduction

A good semantics that supports all phases from programming, compiling, verification, and optimization to execution is indispensable for the reliable application of a programming language. Extended lambda calculi are widely used to provide operational semantics for programming languages, e.g. the semantics of non-strict functional programming languages like Haskell [18] and Clean [19] can be defined by a small-step (rewriting) reduction in a lambda calculus. An efficient evaluation of programs in these languages is based on call-by-need evaluation that implements call-by-name evaluation by exploiting sharing of subexpressions in order to avoid multiple evaluation of the same subexpression. Hence it is important to investigate lambda calculi having a possibility to represent sharing of subexpressions, which is usually made explicit by let-expressions or by recursive let-expressions [5,2,6,4,14].

Reasoning about the semantics requires a notion of equality. Conversion equality is the classic variant, which is known to be inadequate, since not all useful equations can be justified. Defining equality as observational equality, also known as contextual equality, regards expressions as equal, if they cannot be distinguished by all permitted observations, where contextual equality defines as observation the convergence of $C[s]$ for any context C , i.e. s, t are observationally equivalent, if for all contexts C : $C[s]$ converges iff $C[t]$ converges. This is also the coarsest equality for this observation, and justifies correctness of a maximal number of program transformations.

1.1 Call-by-Name, Call-by-Need and Lambda-Calculi with Let or Letrec

An early and influential comparison between different implementations of lambda-calculi was Plotkin's [20] treatment of call-by-value, call-by-name strategies and different abstract machines as implementations, where Plotkin used, besides conversion, also contextual equivalence for comparing strategies and expressions. One result in [20] is that call-by-value and call-by-name are essentially different in the considered lambda calculi. Comparing these strategies with call-by-need leads as a natural approach to extending the lambda-calculus syntax by **let** or **letrec**. It is well-known that non-recursive let-expressions can be simulated by an application (see e.g. [5]). It is also well-known that **letrec** has improved sharing properties during reductions (see e.g. [212]), even better than an encoding using the fixpoint combinator Y , and also allows in several cases to syntactically detect non-termination during evaluation.

In calculi with sharing an important issue is in which cases an improvement of sharing is permitted, or the contrary, which kind of unsharing is permitted, perhaps to enable other program transformations. Note that in non-deterministic calculi, arbitrarily modifying sharing is in general not correct, but correct in special cases (see e.g. [1617]). There are also undecided cases, for which the issue of correctness is unsolved, see the letrec-calculi in [178]. In the deterministic **letrec**-calculus treated in [24], the copy reduction is proved only correct if the expression is not copied into an abstraction. The technical problem of showing correctness of the general copy-transformation is that proofs based on diagrams or rearranging the reduction do not work. Even the proofs that the restricted copy-reduction, where only abstractions or variables are allowed to be copied, is intricate and requires splitting the reduction and a complex measure on reduction sequences [2421].

There are several papers investigating the relationship between call-by-name and call-by-need calculi (see e.g. [5414]). Other work on lambda-calculi extended with letrec is centered around confluence, non-confluence or variants of confluence of the reduction relation of the calculi [623]. A proof of the observational equivalence of a call-by-name and a call-by-need calculus with non-recursive let is in [14], which also mentions at the very end an open question, which can be reformulated as the question, whether a letrec-calculus with a reduction that allows only to copy values is strong enough to show also that

copying arbitrary expressions is correct. A similar question is also implicitly mentioned as unresolved in [4]. As far as we know, there is no proof for this equality w.r.t. contextual equivalence for a calculus using recursive let and call-by-need reductions.

The paper [10] provides a fully abstract denotational semantics for a deterministic extended lambda-calculus with letrec, and a “referential transparency” property is proved. This means for a small calculus similar to LR λ , that the copy-transformation is correct w.r.t. contextual equality. The correctness of the copy transformation for LR λ could be derived from this result, however, the term representation in [10] presupposes the correctness of the transformation $C[s] \rightarrow (\text{letrec } x = s \text{ in } C[x])$, which would require a correctness proof in LR λ , and moreover, the denotational method does not support the comparison of the different evaluation strategies. Also, it is not possible via using the result in [10] to resolve the open problem in [14,4], since these concern evaluation strategies. It is also unclear whether Jeffrey’s denotational proofs can be used for a calculus with case and constructors, since confluence does no longer hold (see [6]), but his method is based on confluence properties; it is also far from obvious how his methods could be adapted to non-deterministic call-by-need calculi with letrec.

The work on letrec-calculi in [2] proves an equivalence of call-by-name and call-by-need, however, for a non-maximal equivalence, i.e. one that distinguishes more expressions than contextual equivalence, a corresponding example can be found in [9]: there are two contextually equivalent lambda terms, $\lambda x.(x x) \sim_c \lambda x.x (\lambda y.x y)$, which have different Böhm-trees, and also different Levy-Longo-trees. However, these terms are contextually equivalent in our calculus.

1.2 Structure and Result of This Paper

This paper demonstrates a new proof method by treating a tiny letrec-calculus LR λ which is equipped with a normal order reduction and a contextual semantics as definition of equality of expressions. First it defines the infinite trees corresponding to the unrolling of expressions as in the 111-calculus of [11]. Then reduction on the infinite trees is defined, where the basic rule is the beta-rule, and the other rule $\overset{\infty}{\rightarrow}$ is a generalization of the (parallel) 1-reduction (see [7]); it can also be seen as an infinite development (see also [11]). It is shown that convergence of expressions in the call-by-need lambda-calculus, as well as for the call-by-name calculus is equivalent to convergence of beta-reduction on the corresponding infinite trees. An essential step is the standardization lemma for $\overset{\infty,*}{\rightarrow}$ -reductions. Finally, as a corollary we obtain the correctness of the general copy-rule in LR λ (see Theorem 4.10). The equivalence of the call-by-need letrec-calculus LR λ with its call-by-name variant is proved in Theorem 5.8, which solved an open problem mentioned in [4].

It is also shown as a spin-off that (cp) and (lll) are correct (see Theorems 4.10, 4.11); the proof of correctness of (lbeta) is omitted, but can be done by copying the proofs in [24]. Our results imply that the calculus LR λ together with its contextual equivalence is equivalent to the theory of the lazy lambda-calculus [1].

As a summary, we have demonstrated that going via a calculus on infinite trees is a successful and extendible method to resolve questions concerning correctness of copy-related transformations in call-by-need letrec-calculi. We checked that our purely operational method can be adapted to the extensions of the calculi by case and constructors [23], and are confident that also for an extension by non-deterministic operators it is possible to prove correctness of copy-related transformations, for which currently there is no other proof method.

2 Syntax and Reductions of the Functional Core Language LR λ

2.1 The Language and the Reduction Rules

We define the calculus LR λ consisting of a language $\mathcal{L}(\text{LR}\lambda)$ and its reduction rules, presented in this section, the normal order reduction strategy and contextual equivalence. The syntax for expressions E is as follows:

$$E ::= V \mid (E_1 E_2) \mid (\lambda V.E) \mid (\mathbf{letrec} \ V_1 = E_1, \dots, V_n = E_n \ \mathbf{in} \ E)$$

where E, E_i are expressions and V, V_i are variables. The expressions $(E_1 E_2)$, $(\lambda V.E)$, $(\mathbf{letrec} \ V_1 = E_1, \dots, V_n = E_n \ \mathbf{in} \ E)$ are called *application*, *abstraction*, or *letrec-expression*, respectively.

All **letrec**-expressions obey the following conditions: The variables V_i in the bindings are all distinct. We also assume that the bindings in **letrec** are commutative, i.e. **letrec**s with bindings interchanged are considered to be syntactically equivalent. The bindings in **letrec** are recursive: I.e., the scope of x_j in $(\mathbf{letrec} \ x_1 = E_1, \dots, x_j = E_j, \dots, x_n = E_n \ \mathbf{in} \ E)$ is E and all expressions E_i for $i = 1, \dots, n$. This fixes the notions of closed, open expressions and α -renamings. Free and bound variables in expressions are defined using the usual conventions. Variable binding primitives are λ and **letrec**. The set of free variables in an expression t is denoted as $FV(t)$. We will use positions in the Dewey decimal notation to speak about tree addresses. For simplicity we use the distinct variable convention: I.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly rename bound variables in the result by α -renaming if necessary to obey this convention. Note that this is only necessary for the copy rule (cp) (see below). We omit parentheses in nested applications: $(s_1 \dots s_n)$ denotes $(\dots (s_1 s_2) \dots s_n)$.

Sometimes we abbreviate the notation of **letrec**-expression $(\mathbf{letrec} \ x_1 = E_1, \dots, x_n = E_n \ \mathbf{in} \ E)$, as $(\mathbf{letrec} \ Env \ \mathbf{in} \ E)$, where $Env \equiv \{x_1 = E_1, \dots, x_n = E_n\}$. This will also be used freely for parts of the bindings. The set of variables bound in an environment Env is denoted as $LV(Env)$.

In the following we define different context classes and contexts. To visually distinguish context classes from individual contexts, we use different text styles. The class \mathcal{C} of all *contexts* is the set of all expressions C from LR λ , where the symbol $[\cdot]$, the *hole*, is a predefined context that is syntactically treated as an atomic expression, such that $[\cdot]$ occurs exactly once in C . Given a term t and a

(lbeta)	$((\lambda x.s) r) \rightarrow (\text{letrec } x = r \text{ in } s)$
(cp-in)	$(\text{letrec } x = s, Env \text{ in } C[x]) \rightarrow (\text{letrec } x = s, Env \text{ in } C[s])$ where s is an abstraction or a variable
(cp-e)	$(\text{letrec } x = s, Env, y = C[x] \text{ in } r) \rightarrow (\text{letrec } x = s, Env, y = C[s] \text{ in } r)$ where s is an abstraction or a variable
(llet-in)	$(\text{letrec } Env_1 \text{ in } (\text{letrec } Env_2 \text{ in } r))$ $\rightarrow (\text{letrec } Env_1, Env_2 \text{ in } r)$
(llet-e)	$(\text{letrec } Env_1, x = (\text{letrec } Env_2 \text{ in } s_x) \text{ in } r)$ $\rightarrow (\text{letrec } Env_1, Env_2, x = s_x \text{ in } r)$
(lapp)	$((\text{letrec } Env \text{ in } t) s) \rightarrow (\text{letrec } Env \text{ in } (t s))$

Fig. 1. Reduction Rules for Call-By-Need

context C , we will write $C[t]$ for the expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t , where this replacement is meant syntactically, i.e., a variable capture is permitted.

Definition 2.1. *A value is an abstraction. We denote values by the letters v, w . A weak head normal form (WHNF) is either a value, or an expression $(\text{letrec } Env \text{ in } v)$, where v is a value.*

The reduction rules in figure 1 are defined more liberally than necessary for the normal order reduction, in order to permit an easy use as transformations.

Definition 2.2 (Reduction Rules of the Calculus LR λ). *The (base) reduction rules for the calculus and language LR λ are defined in figure 1. The union of (llet-in) and (llet-e) is called (llet), the union of (cp-in) and (cp-e) is called (cp), and the union of (llet) and (lapp) is called (ll). Note that in the rule (cp-e) the variables x, y may be equal, but if (cp-e) is used as a normal-order reduction step (see below), then $x \neq y$.*

Reductions (and transformations) are denoted using an arrow with superscripts, e.g. \xrightarrow{llet} . To explicitly state the context in which a particular reduction is executed we annotate the reduction arrow with the context in which the reduction takes place. If no confusion arises, we omit the context at the arrow. The redex of a reduction is the term as given on the left side of a reduction rule. Transitive closure of reductions is denoted by a $+$, reflexive transitive closure by a $$. E.g. $\xrightarrow{*}$ is the reflexive, transitive closure of \rightarrow .*

2.2 The Unwind Algorithm

The following labeling algorithm (UNWIND) will detect the position to which a reduction rule will be applied according to normal order. It uses three labels: S, T, V , where T means reduction of the top term, S means reduction of a sub-term, V labels already visited subexpressions, and $S \vee T$ matches T as well as S . The algorithm does not look into S -labeled letrec-expressions. We also denote the fresh V only in the result of the UNWIND-steps, and do not indicate the

(lbeta)	$C[((\lambda x.s)^S r)] \rightarrow C[(\mathbf{letrec} \ x = r \ \mathbf{in} \ s)]$
(cp-in)	$(\mathbf{letrec} \ x = s^S, \mathit{Env} \ \mathbf{in} \ C[x^V]) \rightarrow (\mathbf{letrec} \ x = s, \mathit{Env} \ \mathbf{in} \ C[s])$ where s is an abstraction or a variable
(cp-e)	$(\mathbf{letrec} \ x = s^S, \mathit{Env}, y = C[x^V] \ \mathbf{in} \ r) \rightarrow (\mathbf{letrec} \ x = s, \mathit{Env}, y = C[s] \ \mathbf{in} \ r)$ where s is an abstraction or a variable
(llet-in)	$(\mathbf{letrec} \ \mathit{Env}_1 \ \mathbf{in} \ (\mathbf{letrec} \ \mathit{Env}_2 \ \mathbf{in} \ r)^S)$ $\rightarrow (\mathbf{letrec} \ \mathit{Env}_1, \mathit{Env}_2 \ \mathbf{in} \ r)$
(llet-e)	$(\mathbf{letrec} \ \mathit{Env}_1, x = (\mathbf{letrec} \ \mathit{Env}_2 \ \mathbf{in} \ s_x)^S \ \mathbf{in} \ r)$ $\rightarrow (\mathbf{letrec} \ \mathit{Env}_1, \mathit{Env}_2, x = s_x \ \mathbf{in} \ r)$
(lapp)	$C[(\mathbf{letrec} \ \mathit{Env} \ \mathbf{in} \ t)^S s] \rightarrow C[(\mathbf{letrec} \ \mathit{Env} \ \mathbf{in} \ (t \ s))]$

Fig. 2. Normal Order Reduction Rules

already existing V -labels. For a term s the labeling algorithm starts with s^T , where no subexpression in s is labeled. The rules of the labeling algorithm are:

$$\begin{array}{ll}
(\mathbf{letrec} \ \mathit{Env} \ \mathbf{in} \ t)^T & \rightarrow (\mathbf{letrec} \ \mathit{Env} \ \mathbf{in} \ t^S)^V \\
(s \ t)^{S \vee T} & \rightarrow (s^S \ t)^V \\
(\mathbf{letrec} \ x = s, \mathit{Env} \ \mathbf{in} \ C[x^S]) & \rightarrow (\mathbf{letrec} \ x = s^S, \mathit{Env} \ \mathbf{in} \ C[x^V]) \\
& \text{if } s \text{ was not labeled} \\
(\mathbf{letrec} \ x = s, y = C[x^S], \mathit{Env} \ \mathbf{in} \ t) & \rightarrow (\mathbf{letrec} \ x = s^S, y = C[x^V], \mathit{Env} \ \mathbf{in} \ t) \\
& \text{if } s \text{ was not labeled and if } C[x] \neq x
\end{array}$$

If UNWIND tries to label an already labelled subterm, then it fails. Otherwise, and if no more rule is applicable, it succeeds. In any case, UNWIND terminates. For example for $(\mathbf{letrec} \ x = x \ \mathbf{in} \ x)^T$ it will stop with $(\mathbf{letrec} \ x = x^S \ \mathbf{in} \ x^V)^V$.

Definition 2.3 (Normal Order Reduction). *A normal order reduction is defined as the reduction at the position of the final label S , or one position higher up, or copying the term from the final position to the position before, as indicated in figure 2. A normal-order reduction step is denoted as \xrightarrow{n} . Note that normal order reduction is unique.*

Definition 2.4. *A normal order reduction sequence is called an (normal-order) evaluation if the last term is a WHNF. Otherwise, i.e. if the normal order reduction sequence is non-terminating, or if the last term is not a WHNF, but has no further normal order reduction, then we say that it is a failing normal order reduction sequence.*

For a term t , we write $t \Downarrow$ iff there is an evaluation starting from t . We call this the evaluation of t . If $t \Downarrow$, we also say that t is converging (or terminating). Otherwise, if there is no evaluation of t , we write $t \not\Downarrow$.

Definition 2.5 (contextual preorder and equivalence). *Let s, t be terms. Then:*

$$\begin{array}{l}
s \leq_c t \text{ iff } \forall C[\cdot] : C[s] \Downarrow \Rightarrow C[t] \Downarrow \\
s \sim_c t \text{ iff } s \leq_c t \wedge t \leq_c s
\end{array}$$

3 Reductions on Trees

In the following we use “expression” for finite expressions including `letrec`, and “tree” for the finite or infinite trees, which are only built from applications, abstractions and variables.

The infinite tree corresponding to an expression is intended to be the letrec-unfolding of the expression with the extra condition that cyclic variable chains lead to local nontermination, represented by the symbol \perp . This corresponds to the infinite trees in the 111-variant of the calculus in [11]. A rigorous definition is as follows, where we use the explicit binary application operator $@$, since it is easier to explain, but stick to the common notation in examples.

Definition 3.1. *Given an expression t , the infinite tree $IT(t)$ of t is defined by giving an algorithm to compute for every position p the label of the infinite tree at position p , where the algorithm starts with $t|_p$, where by a slight abuse of notation, the notation $r|_p$ is used as a label. The algorithm uses the rules below to move the label $r|_p$ around within t . The computation is successful, iff the label is $r|_\varepsilon$ and the conditions below hold. If the computation does not terminate, then it is not successful, and the result is undefined with the following exception: If the position ε hits the same (let-bound) variable twice, using the rules below, then the result is \perp .*

The computed term-label for the position ε is as follows:

$$\begin{aligned} C[(@ s t)|_\varepsilon] &\mapsto @ \\ C[x|_\varepsilon] &\mapsto x && \text{if } x \text{ is a free or a lambda-bound variable} \\ C[(\lambda x.s)|_\varepsilon] &\mapsto \lambda x \end{aligned}$$

In general, we proceed using the rules below:

$$\begin{aligned} C[(\lambda x.s)|_{1.p}] &\rightarrow C[\lambda x.(s|_p)] \\ C[(@ s t)|_{1.p}] &\rightarrow C[(@ s|_p t)] \\ C[(@ s t)|_{2.p}] &\rightarrow C[(@ s t|_p)] \\ C[(\text{letrec Env in } r)|_p] &\rightarrow C[(\text{letrec Env in } r|_p)] \\ C_1[(\text{letrec } x = s, \text{Env in } C_2[x|_p])] &\rightarrow C_1[(\text{letrec } x = s|_p, \text{Env in } C_2[x])] \\ C_1[(\text{letrec } x = s, y = C_2[x|_p], \text{Env in } r)] &\rightarrow C_1[(\text{letrec } x = s|_p, y = C_2[x], \text{Env in } r)] \end{aligned}$$

The equivalence of trees is syntactic, where α -equal trees are assumed to be equivalent. A tree of the form $\lambda x.s$ is called a value.

Example 3.2. The expression `letrec $x = x, y = (\lambda z.z) x y$ in y` has the corresponding tree $((\lambda z.z) \perp ((\lambda z.z) \perp ((\lambda z.z) \perp \dots)))$.

Definition 3.3. *Reduction contexts \mathcal{R} for (infinite) trees are defined by $\mathcal{R} ::= [\cdot] \mid (@ \mathcal{R} E)$, where E means a tree.*

Lemma 3.4. *Let s, t be expressions and C be a context. Then $IT(s) = IT(t) \Rightarrow IT(C[s]) = IT(C[t])$.*

Lemma 3.5. *Let s, t be expressions and $s \rightarrow t$ by (cp) or (lll). Then $IT(s) = IT(t)$.*

Definition 3.6. *(betaTr) is the only reduction rule on trees. It is allowed in any tree-context.*

$$\boxed{(betaTr) \ ((\lambda x.s) r) \rightarrow s[r/x]}$$

If the reduction rule is applied within an \mathcal{R} -context, we will call it an \mathcal{R} -reduction on trees. A sequence of \mathcal{R} -reductions of T that terminates with a value tree is called evaluation. If T has an evaluation, then we also say T converges and denote this as $T \Downarrow$.

Note that (betaTr) as a reduction may modify infinitely many positions, since there may be infinitely many positions of the variable x . E.g. a top-level (betaTr) of $IT((\lambda x.(\mathbf{letrec} \ z = (z \ x) \ \mathbf{in} \ z)) \ r) = (\lambda x.((\dots (\dots \ x) \ x) \ x)) \ r$ modifies the infinite number of occurrences of x . Further note that (betaTr) does not overlap with itself, where we ignore overlaps within the meta-variables s, r .

Lemma 3.7. *Let s be an expression and let $IT(s)$ be a value tree. Then $s \Downarrow$.*

We will use a variant of infinite outside-in developments [7,11] as a reduction on trees that may reduce infinitely many redexes in one step, which can also be seen as a generalization of the 1-reduction to infinite trees. For a more detailed definition, in particular concerning the labeling, see [22]. The idea is that a single reduction (lbeta) of a single redex in an expression is mirrored as reducing a perhaps infinite set of redexes in the corresponding infinite tree.

Definition 3.8. *For trees S, T , we define the reduction $S \xrightarrow{\infty} T$ as a generalization of the (parallel) 1-reduction (see [7]) as follows. We mark a possibly infinite subset of all (betaTr)-redexes in S , say with a \dagger (the subset may also be empty). The reduction constructs a new infinite tree top-down by iteratedly using labelled reduction, where the label of the redex is removed before the reduction. If the reduction does not terminate for a subtree at its top level, then this subtree is the constant \perp in the result. This recursively defines the result tree top-down. We write $T \Downarrow(\infty)$ if $T \xrightarrow{\infty, *} T'$, where T' is a value tree.*

The reduction $S \xrightarrow{\forall, \infty} T$ is defined as the specific $S \xrightarrow{\infty} T$ -reduction, if all (betaTr)-redexes in S are labeled.

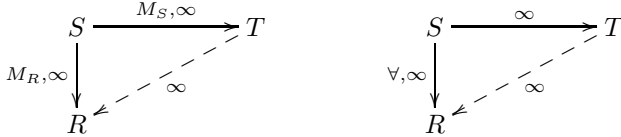
Note that even for a tree with only two marked redexes, it is possible that after the first reduction during construction of the result tree, infinitely many redexes are labeled.

Example 3.9. We give two examples for a $\xrightarrow{\infty}$ -reduction:

- $t = (\lambda z. \mathbf{letrec} \ y = \lambda u.u, x = (z \ (y \ y) \ x) \ \mathbf{in} \ x)$. The infinite tree $IT(t)$ is like an infinite list, descending to the right, with elements $((\lambda u.u) \ \lambda u.u)$. The ∞ -reduction may label any subset of these redexes, even infinitely many, and then reduce them by (betaTr).

- $t = (\mathbf{letrec} \ x = \lambda y.x \ (\lambda u.u) \ \mathbf{in} \ x)$ has the infinite tree $(\lambda y.(\lambda y.(\lambda y.\dots)) (\lambda u.u) (\lambda u.u)) (\lambda u.u)$ which, depending on the labeling, may reduce to itself, or, if all redexes are labeled, it will reduce to \perp , i.e., $t \xrightarrow{\forall, \infty} \perp$.

Lemma 3.10. *For all trees S, R, T : if $S \xrightarrow{\infty} R$ where the set of redex positions is M_R , and $S \xrightarrow{\infty} T$, where the set of redex positions is M_S , and $M_S \subseteq M_R$, then also $T \xrightarrow{\infty} R$. A special case is that $S \xrightarrow{\forall, \infty} R$, and $S \xrightarrow{\infty} T$ imply that $T \xrightarrow{\infty} R$.*



Proof. The argument is that we can mark the (betaTr)-redexes in S that are not reduced in $S \xrightarrow{M_S, \infty} T$. Reduce all M_R -labeled redexes in the reduction $T \xrightarrow{\infty} R$.

In the extended version [22] it is shown:

Theorem 3.11 (Standardization for tree-reduction). *Let S be a tree. Then $S \Downarrow(\infty)$ implies $S \Downarrow$.*

Proof. (very short sketch) The main idea of the proof is that it is sufficient to reduce only a finite number of positions in S by (betaTr) to reach some value. This is done by reorganizing and commuting reduction sequences, until an \mathcal{R} -evaluation is obtained, where reductions that are “too deep” are not performed.

4 Properties of Call-by-Need Convergence

4.1 Call-by-Need Convergence Implies Infinite Tree Convergence

Lemma 4.1. *If $s \xrightarrow{lbeta} t$ for two expressions s, t , then $IT(s) \xrightarrow{\infty} IT(t)$.*

Proof. We label every redex of $IT(s)$ that is derived from the redex corresponding to $s \xrightarrow{lbeta} t$. If the redex in s is $((\lambda x.s') r')$ and s' is not a variable, then the lemma is obvious. The only nontrivial case is that s' is a variable and the subexpression is e.g. of the form $(\mathbf{letrec} \ Env, y_2 = y_1, y_1 = ((\lambda x.y_2) r') \ \mathbf{in} \ s')$, and after the (lbeta)-reduction, and perhaps some (lll)-reductions, y_2 is in a cyclic chain of variables like $(\mathbf{letrec} \ Env, y_2 = y_1, y_1 = y_2, x = r' \ \mathbf{in} \ s')$. In this case the tree-reduction of the redex corresponding to y_1 does not terminate during computing the development, and hence the result will be \perp .

Proposition 4.2. *Let t be an expression. Then $t \Downarrow \Rightarrow IT(t) \Downarrow$.*

Proof. That $IT(t) \Downarrow(\infty)$ holds follows from Lemma 4.1 by induction on the length of evaluation of t , from Lemma 3.5 and from the fact that a WHNF has a value tree as corresponding infinite tree. Then Theorem 3.11 shows that $IT(t) \Downarrow(\infty)$ implies also $IT(t) \Downarrow$.

4.2 Infinite Tree Convergence Implies Call-by-Need Convergence

Now we show the harder part of the desired equivalence in a series of lemmas.

Lemma 4.3. *For every reduction possibility $S_1 \xleftarrow{\mathcal{R}} T \xrightarrow{\infty} S_2$, either $S_1 \xrightarrow{\infty} S_2$ or there is some T' with $S_1 \xrightarrow{\infty} T' \xleftarrow{\mathcal{R}} S_2$. I.e. we have the following forking diagrams for trees between an \mathcal{R} -reduction and an $\xrightarrow{\infty}$ -reduction:*

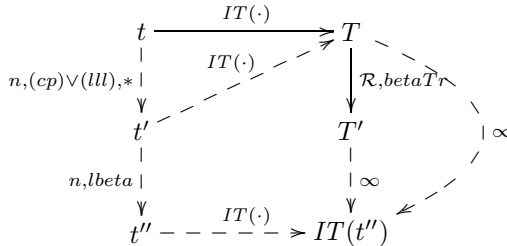


Proof. This follows by checking the overlaps of $\xrightarrow{\infty}$ with \mathcal{R} -reductions.

Lemma 4.4. *Let T be a tree such that there is an \mathcal{R} -evaluation of length n , and let S be a tree with $T \xrightarrow{\infty} S$. Then S has an \mathcal{R} -evaluation of length $\leq n$.*

Proof. Follows from Lemma 4.3 by induction.

Lemma 4.5. *Let t be a term and let $T := IT(t) \xrightarrow{(\text{betaTr})} T'$ be an \mathcal{R} -reduction. Then there is an expression t' , a reduction $t \xrightarrow{n,*} t'$ using (lll) and (cp)-reductions, an expression t'' with $t' \xrightarrow{n,\text{lbeta}} t''$, such that there is a reduction $T' \xrightarrow{\infty} IT(t'')$.*



Proof. The expressions t', t'' are constructed as follows: t' is the resulting term from a maximal normal-order reduction of t consisting only of (cp) and (lll)-reductions. It is clear that such a sequence of $\xrightarrow{(\text{cp}) \vee (\text{lll}), n}$ -reductions is terminating. Then $IT(t) = IT(t')$ by Lemma 3.5. The unique normal-order (lbeta)-redex in t' must correspond to $T \xrightarrow{\mathcal{R},(\text{betaTr})} T'$ and is used for the reduction $t' \xrightarrow{n,\text{lbeta}} t''$. Note that the (lbeta)-redex in t' may correspond to infinitely many redexes in T . Lemma 4.1 shows that there is a reduction $T \xrightarrow{\infty} IT(t'')$, and Lemma 3.10 shows that also $T' \xrightarrow{\infty} IT(t'')$.

Proposition 4.6. *Let t be an expression such that $IT(t) \Downarrow$. Then $t \Downarrow$.*

Proof. The precondition $IT(t) \Downarrow$ implies that there is an \mathcal{R} -evaluation of $IT(t)$ to a value tree. The base case, where no \mathcal{R} -reductions are necessary is treated

in Lemma 3.7. In the general case, let $T \xrightarrow{(betaTr)} T'$ be the unique first \mathcal{R} -reduction of a single redex. Lemma 4.5 shows that there are expressions t', t'' with $t \xrightarrow{n,(cp)\vee(ill),*} t' \xrightarrow{n,lbeta} t''$, and $T' \xrightarrow{\infty} IT(t'')$. Lemma 4.4 shows that the number of \mathcal{R} -reductions of $IT(t'')$ to a value tree is strictly smaller than the number of \mathcal{R} -reductions of T to a value. Hence we can use induction on this length and obtain a normal-order reduction of t to a WHNF.

Convergence is equivalent for a term and its corresponding infinite tree:

Theorem 4.7. *Let t be an expression. Then $t \Downarrow$ iff $IT(t) \Downarrow$.*

Proof. This follows from Propositions 4.2 and 4.6.

Definition 4.8. *Let the generalized copy rule be:*

$$(gcp) \quad C_1[\mathbf{letrec} \ x = r \dots C_2[x] \dots] \rightarrow C_1[\mathbf{letrec} \ x = r \dots C_2[r] \dots]$$

This is just like the rule (cp), but all kinds of terms r can be copied, not only abstractions. Obviously the following holds:

Lemma 4.9. *If $s \xrightarrow{gcp} t$, then $IT(s) = IT(t)$*

Theorem 4.10. *Let s, t be expressions with $s \xrightarrow{gcp} t$. Then $s \sim_c t$.*

Proof. Lemma 3.4 shows that it is sufficient to show equivalence of termination of s, t . Lemma 4.9 implies $IT(s) = IT(t)$. Hence equivalence of termination follows from Theorem 4.7.

Theorem 4.11. *Let s, t be expressions with $s \xrightarrow{ill} t$. Then $s \sim_c t$.*

Proof. Follows in the same way as in the proof of Theorem 4.10 using Lemma 3.5.

5 Relation Between Call-by-Name and Call-by-Need

For the same language we now treat the call-by-name variant of the reduction strategy using beta-reduction instead of the rule (lbeta) that respects sharing.

Definition 5.1. *The call-by-name normal-order reduction is defined by replacing the (lbeta)-reduction in the call-by-need normal-order reduction by (beta):*

$$(beta) \quad ((\lambda x.s) r) \rightarrow s[r/x]$$

where the same redex is used. We denote the reduction as \xrightarrow{name} , the corresponding call-by-name convergence of a term t as $t \Downarrow(name)$, and the corresponding contextual preorder and equivalence as $\leq_{c,name}$ and $\sim_{c,name}$, respectively.

Note that for $a \in \{(ill), (cp)\}$ the relation $\xrightarrow{n,a}$ is the same as $\xrightarrow{name,a}$. However, we could also define other call-by-name-variants with unrestricted copy.

We give an example showing that the call-by-name evaluation and the call-by-need evaluation may have essentially different infinite tree evaluations.

Example 5.2. We start with the term $(\mathbf{letrec} z = (\lambda x.(\lambda y.x)) (z z) \mathbf{in} z z)$. The call-by-need normal order reduction is as follows:

$$\begin{aligned}
& \xrightarrow{\text{lbeta}} (\mathbf{letrec} z = (\mathbf{letrec} x = z z \mathbf{in} \lambda y.x) \mathbf{in} z z) \\
& \xrightarrow{\text{ll}} (\mathbf{letrec} z = \lambda y.x, x = z z \mathbf{in} z z) \\
& \xrightarrow{\text{cp}} (\mathbf{letrec} z = \lambda y.x, x = z z \mathbf{in} (\lambda y.x) z) \\
& \xrightarrow{\text{lbeta}} (\mathbf{letrec} z = \lambda y.x, x = z z \mathbf{in} (\mathbf{letrec} y = z \mathbf{in} x)) \\
& \xrightarrow{\text{ll}} (\mathbf{letrec} z = \lambda y.x, x = z z, y = z \mathbf{in} x) \\
& \xrightarrow{\text{cp}} (\mathbf{letrec} z = \lambda y.x, x = (\lambda y'.x) z, y = z \mathbf{in} x) \\
& \xrightarrow{\text{lbeta}} (\mathbf{letrec} z = \lambda y.x, x = (\mathbf{letrec} y' = z \mathbf{in} x), y = z \mathbf{in} x) \\
& \xrightarrow{\text{ll}} (\mathbf{letrec} z = \lambda y.x, x = x, y' = z, y = z \mathbf{in} x)
\end{aligned}$$

Thus it fails. The call-by-name normal order reduction loops, where the first reduction gives $(\mathbf{letrec} z = (\lambda y.(z z)) \mathbf{in} z z)$, which immediately starts a loop using (beta) and (cp)-reductions.

Thus the call-by-name and call-by-need reductions have a different trace of infinite trees, hence an easy correspondence proof of the reductions is not possible. Witnesses are the expressions $s_1 = (\mathbf{letrec} z = (\lambda y.(z z)) \mathbf{in} z z)$ and $s_2 = (\mathbf{letrec} z = \lambda y.x, x = (\lambda y'.x) z, y = z \mathbf{in} x)$ that have the same infinite tree, and the call-by-name reduction of s_1 gives an expression with the same infinite tree, whereas the call-by-need reduction of s_2 results in the tree \perp .

5.1 Call-by-Name Convergence Implies Infinite Tree Convergence

Lemma 5.3. *If $s \xrightarrow{\text{beta}} t$ for two expressions s, t , then $IT(s) \xrightarrow{\infty} IT(t)$.*

Proof. Easy, since $\xrightarrow{\text{beta}}$ and $\xrightarrow{\text{lbeta}}$ result in the same tree.

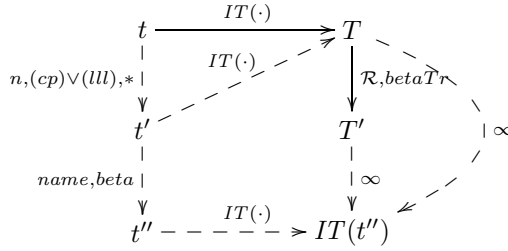
Proposition 5.4. *Let t be an expression. Then $t \Downarrow(\text{name}) \Rightarrow IT(t) \Downarrow$.*

Proof. This follows from Lemma 5.3 by induction on the length of the call-by-name evaluation of t , from Lemma 3.5 using the standardization theorem 3.11 and from the fact that a WHNF has a value tree as corresponding infinite tree.

5.2 Infinite Tree Convergence Implies Call-by-Name Convergence

Now we show the desired implication also for call-by-name.

Lemma 5.5. *Let t be a term and let $T := IT(t) \xrightarrow{(\text{betaTr})} T'$ be an \mathcal{R} -reduction. Then there is an expression t' , a reduction $t \xrightarrow{n,*} t'$ using (ll) and (cp)-reductions, an expression t'' with $t' \xrightarrow{\text{name,beta}} t''$, such that there is a reduction $T' \xrightarrow{\infty} IT(t'')$.*



Proof. The expressions t', t'' are constructed as follows: t' is the resulting term from a maximal normal-order reduction consisting only of (cp) and (ll)-reductions. It is clear that such a sequence of $\xrightarrow{(cp)\vee(ll),n}$ -reductions is terminating. Then $IT(t) = IT(t')$ by Lemma 3.5. The unique normal-order (beta)-redex in t' corresponding to $T \xrightarrow{(betaTr)} T'$ is used for the reduction $t' \xrightarrow{name,beta} t''$. Note that the (beta)-redex in t' may correspond to infinitely many redexes in T . Lemma 5.3 shows that there is a reduction $T \xrightarrow{\infty} IT(t'')$, and Lemma 3.10 shows that also $T' \xrightarrow{\infty} IT(t'')$.

Proposition 5.6. *Let t be an expression such that $IT(t)\Downarrow$. Then $t\Downarrow(name)$.*

Proof. The precondition $IT(t)\Downarrow$ means that there is an \mathcal{R} -evaluation of $T := IT(t)$ to a value tree. The base case, where no \mathcal{R} -reductions are necessary is treated in Lemma 3.7. In the general case, let $T \xrightarrow{(betaTr)} T'$ be the unique first \mathcal{R} -reduction of a single redex. Lemma 5.5 shows that there are expressions t', t'' with $t \xrightarrow{n,(cp)\vee(ll),*} t' \xrightarrow{name,beta} t''$, and $T' \xrightarrow{\infty} IT(t'')$. Lemma 4.4 shows that the number of \mathcal{R} -reductions of $IT(t'')$ to a value tree is strictly smaller than the number of \mathcal{R} -reductions of T to a value. Hence we can use induction on this length and obtain a call-by-name normal-order reduction of t to a WHNF.

Now we can show that call-by-name termination for a term is equivalent to convergence of its corresponding infinite tree.

Theorem 5.7. *Let t be an expression. Then $t\Downarrow(name)$ iff $IT(t)\Downarrow$.*

Proof. Follows from Propositions 5.4 and 5.6.

The strategies call-by-need and call-by-name are equivalent:

Theorem 5.8. *The contextual preorders for call-by-need and call-by-name are equivalent.*

Proof. This follows from Theorems 4.7 and 5.7.

5.3 Relation to the Lazy Lambda Calculus

Without explicit proof, let us remark that our results imply that the calculus LR λ together with its contextual equivalence is equivalent to the theory of the

lazy lambda-calculus [1]: The lazy lambda calculus as presented in [1] is an untyped lambda calculus with a normal order reduction, abstractions as the outcome of reductions, and it is equipped with a contextual equivalence. Our result shows that w.r.t. contextual equivalence, the letrec-expressions can be expressed using the fixpoint operator Y , and since also the contexts correspond, there is one-to-one correspondence between the expressions that respects the contextual preorders in the respective calculi.

6 Conclusion

We demonstrated the proof method via infinite trees by showing correctness of an unrestricted copy-reduction and the equivalence of call-by-name and call-by-need for a tiny deterministic letrec-calculus $LR\lambda$. We have checked that the method also works in letrec-calculi extended by constructors and case-expressions [23], and are sure that it can be applied to the letrec-calculi from [5,26,41,4], if contextual equivalence as equality is adopted, which appears to cover all the desired equalities in these calculi. It could also be applied to the record calculus in [13] after specializing meaning-preservation to contextual equivalence. This shows that the proof method using infinite trees that we have successfully applied has a great potential in exhibiting correctness of variants of copy-transformations in different kinds of calculi with cyclic sharing mechanisms.

For non-deterministic calculi like [16,15,21] we plan to extend the method to show correctness of the copy-reduction for deterministic subexpressions, which appears to be a hard obstacle for other methods.

Acknowledgements

I thank David Sabel for reading and correcting drafts of this paper. I also acknowledge discussions with Elena Machkasova.

References

1. Abramsky, S.: The lazy lambda calculus. In: Turner, D. (ed.) *Research Topics in Functional Programming*, pp. 65–116. Addison-Wesley, Reading (1990)
2. Ariola, Z.M., Blom, S.: Cyclic lambda calculi. In: *TACS*, pp. 77–106, Sendai, Japan (1997)
3. Ariola, Z.M., Blom, S.: Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic* 117, 95–168 (2002)
4. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. *J. functional programming* 7(3), 265–301 (1997)
5. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: *A call-by-need lambda calculus*. In: *Principles of Programming Languages*, San Francisco, California, pp. 233–246. ACM Press, New York (1995)
6. Ariola, Z.M., Klop, J.W.: Lambda calculus with explicit recursion. *Information and Computation* 139(2), 154–233 (1997)

7. Barendregt, H.P.: *The Lambda Calculus. Its Syntax and Semantics*. North-Holland, Amsterdam, New York (1984)
8. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Thiagarajan, P.S., Yap, R. (eds.) *ASIAN 1999*. LNCS, vol. 1742, pp. 62–73. Springer, Heidelberg (1999)
9. Dezani-Ciancaglini, M., Tiuryn, J., Urzyczyn, P.: Discrimination by parallel observers: The algorithm. *Information and Computation* 150(2), 153–186 (1999)
10. Jeffrey, A.: A fully abstract semantics for concurrent graph reduction. In: *Proc. LICS*, pp. 82–91 (1994)
11. Kennaway, R., Klop, J.W., Sleep, M.R., de Vries, F.-J.: Infinitary lambda calculus. *Theor. Comput. Sci.* 175(1), 93–125 (1997)
12. Launchbury, J.: A natural semantics for lazy evaluation. In: *Proc. 20th Principles of Programming Languages* (1993)
13. Machkasova, E., Turbak, F.A.: A calculus for link-time compilation. In: *ESOP'2000* (2000)
14. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. *J. Functional programming* 8, 275–317 (1998)
15. Moran, A.K.D.: Call-by-name, call-by-need, and McCarthys Amb. PhD thesis, Dept. of Comp. Science, Chalmers University, Sweden (1998)
16. Moran, A.K.D., Sands, D., Carlsson, M.: Erratic fudgets: A semantic theory for an embedded coordination language. In: Ciancarini, P., Wolf, A.L. (eds.) *COORDINATION 1999*. LNCS, vol. 1594, pp. 85–102. Springer, Heidelberg (1999)
17. Moran, A.K.D., Sands, D., Carlsson, M.: Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.* 46(1-2), 99–135 (2003)
18. Peyton Jones, S.: *Haskell 98 Language and Libraries*. Cambridge University Press (2003) www.haskell.org
19. Plasmeijer, R., van Eekelen, M.: The concurrent Clean language report: Version 1.3 and 2.0. Technical report, Dept. of Computer Science, University of Nijmegen (2003) <http://www.cs.kun.nl/clean/>
20. Plotkin, G.D.: Call-by-name, call-by-value, and the lambda-calculus. *Theoretical Computer Science* 1, 125–159 (1975)
21. Sabel, D., Schmidt-Schauß, M.: A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. Frank report 24, Inst. Informatik. J.W.G.-university Frankfurt (2006)
22. Schmidt-Schauß, M.: Equivalence of call-by-name and call-by-need for lambda-calculi with letrec. Frank report 25, Inst. Informatik. J.W.G.-university Frankfurt (2006)
23. Schmidt-Schauß, M.: Correctness of copy in calculi with letrec, case and constructors. Frank report 28, Inst. Informatik. J.W.G.-University Frankfurt (2007)
24. Schmidt-Schauß, M., Schütz, M., Sabel, D.: A complete proof of the safety of Nöcker's strictness analysis. Frank report 20, Inst. Informatik. J.W.G.-University Frankfurt (2005)

A Characterization of Medial as Rewriting Rule

Lutz Straßburger

INRIA Futurs, Projet Parsifal

École Polytechnique — LIX — Rue de Saclay — 91128 Palaiseau Cedex — France
<http://www.lix.polytechnique.fr/~lutz>

Abstract. Medial is an inference rule scheme that appears in various deductive systems based on deep inference. In this paper we investigate the properties of medial as rewriting rule independently from logic. We present a graph theoretical criterion for checking whether there exists a medial rewriting path between two formulas. Finally, we return to logic and apply our criterion for giving a combinatorial proof for a decomposition theorem, i.e., proof theoretical statement about syntax.

1 Introduction

An interesting question to ask about a given rewriting system is not only whether it is terminating or confluent, but also whether there is a rewriting path between two given terms. This question occurs, for example, in proof search, where one is interested in finding a proof for a formula P , i.e., a rewriting path from “truth” to P using the inference rules of the deductive system. Alternatively, one can ask for a refutation of P , which is nothing but a rewriting path from P to “falsum”, where the meanings of “truth” and “falsum” depend on the logic in question.

The next natural question to ask is whether we can characterize the existence of a rewriting path between two given terms independently from the rewriting system. For example in [BdGR97], a rewriting system was presented which could be characterized by the inclusion relation of series-parallel orders. Other well-known examples of such characterizations are the various correctness criteria for proof nets for multiplicative linear logic (e.g., [DR89, Ret96, DHPP99, Str03a]).

The work presented in this paper is in line with these results. The rewriting system that we analyze consists only of the medial rule [BT01], which plays an increasing role in the proof theory for classical propositional logic, in particular, in the investigation of the identity of proofs [Str05] and for giving semantics to proofs [Lam06]. Our characterization will be carried out in terms of *relation webs* [Gug07], and is in spirit very close to the work in [BdGR97].

This paper is organized as follows: In the next section we will first explain informally what the medial rule is. Then, in Sections 3 and 4, we will set the stage by formally defining our rewrite system and by introducing the notion of relation web. The main part of the paper is Section 5, in which we prove our main result. The remaining sections compare the result to related work and show an application in proof theory.

2 What is Medial ?

Let \bullet and \circ be two binary operations and consider the equation

$$(x \bullet y) \circ (w \bullet z) = (x \circ w) \bullet (y \circ z) \quad , \tag{1}$$

which is known under the name “middle four exchange” [Mac71]. If we consider \bullet as “horizontal” composition and \circ as “vertical” composition, we can give (1) the following geometric interpretation:

$$\begin{array}{|c|c|} \hline x & y \\ \hline w & z \\ \hline \end{array} = \begin{array}{|c|c|} \hline x & y \\ \hline w & z \\ \hline \end{array} = \begin{array}{|c|} \hline x \\ \hline w \\ \hline \end{array} \begin{array}{|c|} \hline y \\ \hline z \\ \hline \end{array}$$

Let us now assume that one of \bullet and \circ is stronger, in the sense that the equation (1) gets a direction and becomes a rewriting rule

$$(x \bullet y) \circ (w \bullet z) \rightarrow (x \circ w) \bullet (y \circ z) \quad . \tag{2}$$

If we read \bullet as “and” \wedge and \circ as “or” \vee , then (2) becomes a valid implication of Boolean logic

$$(x \wedge y) \vee (w \wedge z) \rightarrow (x \vee w) \wedge (y \vee z) \quad . \tag{3}$$

while the other direction would not yield a valid implication. The same situation appears in linear logic if we let $\langle \bullet, \circ \rangle$ be any of the pairs $\langle \otimes, \oplus \rangle$, $\langle \wp, \oplus \rangle$, $\langle \&, \oplus \rangle$, $\langle \&, \wp \rangle$, or $\langle \&, \otimes \rangle$. In [BT01], the implication (3) is used as an inference rule in a deductive system for classical logic

$$\text{m} \frac{F\{(A \wedge C) \vee (B \wedge D)\}}{F\{(A \vee B) \wedge (C \vee D)\}} \quad , \tag{4}$$

where A, B, C, D stand for arbitrary formulas and $F\{ \}$ for an arbitrary (positive) formula context. Note that (3) and (4) are just different ways of writing the same thing. In [BT01], Brünnler and Tiu gave the name *medial* to the rule (4). They observed that under the presence of the medial rule, the general contraction rule can be reduced to an atomic version:

$$\text{c} \frac{F\{A \vee A\}}{F\{A\}} \quad \rightsquigarrow \quad \text{c} \frac{F\{a \vee a\}}{F\{a\}} \quad , \tag{5}$$

where A is an arbitrary formula and a is just an atom (or literal). In [Str02], the same has been observed for linear logic.

3 Rewriting with Medial

We have in our language two binary function symbols and a countable set $\mathcal{A} = \{a, b, c, \dots\}$ of constant symbols. The set \mathcal{T} of terms is defined by the grammar

$$\mathcal{T} ::= \mathcal{A} \mid (\mathcal{T} \bullet \mathcal{T}) \mid [\mathcal{T} \circ \mathcal{T}]$$

For the two binary function symbols we use infix notation. To ease the readability, we use different types of parentheses: (\dots) for the \bullet and $[\dots]$ for the \circ .¹

¹ Note that this goes in line with the usual notation used in the literature on deep inference, e.g., [BT01, GS01, DG04].

We will use capital Latin letters to denote terms. To ease readability, we will sometimes write $(x \bullet y \bullet z)$ for $((x \bullet y) \bullet z)$ and $[x \circ y \circ z]$ for $[[x \circ y] \circ z]$.

Let AC be the following set of equations on terms, saying that \bullet and \circ are both associative and commutative:

$$\begin{aligned} (x \bullet y) &\approx (y \bullet x) & ((x \bullet y) \bullet z) &\approx (x \bullet (y \bullet z)) \\ [x \circ y] &\approx [y \circ x] & [[x \circ y] \circ z] &\approx [x \circ [y \circ z]] \end{aligned} \tag{6}$$

where $x, y,$ and z are variables. Let \approx_{AC} be the equational theory induced by AC, i.e., the smallest congruence relation containing AC.

Now let M be the rewriting system consisting only of the medial rule

$$[(x \bullet y) \circ (w \bullet z)] \rightarrow ([x \circ w] \bullet [y \circ z]) \quad , \tag{7}$$

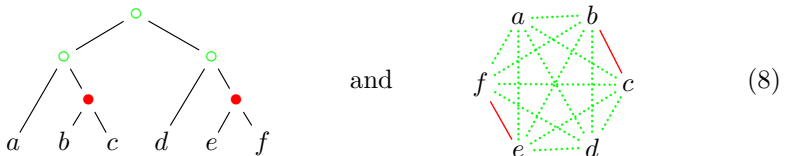
where $x, y, z,$ and w are variables. The object of interest in this paper is the rewrite relation $\rightarrow_{M/AC}$, i.e., rewriting via the medial rule modulo associativity and commutativity of the two binary operations. More formally: Let P and Q be terms. Then $P \rightarrow_{M/AC} Q$, if and only if there are terms P' and Q' such that $P \approx_{AC} P'$ and $P' \rightarrow_M Q'$ and $Q' \approx_{AC} Q$, where $P' \rightarrow_M Q'$ means there is a single rewriting step from P' to Q' using the rule in (7). For more details on the formal definitions see, e.g. [BN98]. Since no ambiguity is possible here, we omit the index AC and simply write $P \approx Q$ instead of $P \approx_{AC} Q$. Further, we write $P \xrightarrow[M]{} Q$ instead of $P \rightarrow_{M/AC} Q$, and we define $\xrightarrow[M]^*$ to be the transitive closure of $\xrightarrow[M]{}$. We are interested in the question: *Under which conditions do we have $P \xrightarrow[M]^* Q$?*

4 Relation Webs

For simplifying the definitions, we will in the following assume that every constant symbol appears at most once in a term. This allows us to ignore the distinction between constants and constant occurrences. What matters in this and the next section are the positions occupied by the constants in the terms.

For a given term P , let \mathcal{V}_P denote the set of constants occurring in P . Let us now treat a term as a binary tree whose inner nodes are labeled by either \bullet or \circ , and whose leaves are the elements of \mathcal{V}_P . For $a, b \in \mathcal{V}_P$ we write $a \overset{\bullet}{P} b$ if their first common ancestor in P is a \bullet and we write $a \overset{\circ}{P} b$ if it is a \circ . Furthermore, we define $\mathcal{E}_P^\bullet = \{(a, b) \in \mathcal{V}_P \times \mathcal{V}_P \mid a \overset{\bullet}{P} b\}$ and $\mathcal{E}_P^\circ = \{(a, b) \in \mathcal{V}_P \times \mathcal{V}_P \mid a \overset{\circ}{P} b\}$. Note that \mathcal{E}_P^\bullet and \mathcal{E}_P° are symmetric, i.e., $(a, b) \in \mathcal{E}_P^\bullet$ iff $(b, a) \in \mathcal{E}_P^\bullet$. We also have $\mathcal{E}_P^\bullet \cap \mathcal{E}_P^\circ = \emptyset$ and $\mathcal{E}_P^\bullet \cup \mathcal{E}_P^\circ = (\mathcal{V}_P \times \mathcal{V}_P) \setminus \{(a, a) \mid a \in \mathcal{V}_P\}$. The triple $\oplus P = \langle \mathcal{V}_P; \mathcal{E}_P^\bullet, \mathcal{E}_P^\circ \rangle$ is called the *relation web of P* . We can think of it as a complete undirected graph with vertices \mathcal{V}_P and edges $\mathcal{E}_P^\bullet \cup \mathcal{E}_P^\circ$ where we color the edges in \mathcal{E}_P^\bullet red and the edges in \mathcal{E}_P° green.

Consider for example the term $P = [[a \circ (b \bullet c)] \circ [d \circ (e \bullet f)]]$. Its syntax tree and its relation web are, respectively,



where the red lines are solid and green lines are drawn as dotted lines.

It is now easy to see that we have the following:

4.1 Proposition. *Let P and Q be terms. Then $\otimes P = \otimes Q$ iff $P \approx Q$.*

More interesting, however, is the question, under which circumstances a triple $\langle \mathcal{V}; \mathcal{E}^\bullet, \mathcal{E}^\circ \rangle$ is indeed the relation web of a term. Let us define a *preweb* to be a triple $\langle \mathcal{V}; \mathcal{E}^\bullet, \mathcal{E}^\circ \rangle$ where \mathcal{E}^\bullet and \mathcal{E}° are symmetric subsets of $\mathcal{V} \times \mathcal{V}$ such that

$$\mathcal{E}^\bullet \cap \mathcal{E}^\circ = \emptyset \quad \text{and} \quad \mathcal{E}^\bullet \cup \mathcal{E}^\circ = (\mathcal{V} \times \mathcal{V}) \setminus \{(a, a) \mid a \in \mathcal{V}\} \quad . \quad (9)$$

4.2 Proposition. *Let $\mathcal{G} = \langle \mathcal{V}; \mathcal{E}^\bullet, \mathcal{E}^\circ \rangle$ be a preweb. Then $\mathcal{G} = \otimes P$ for some term P if and only if we do not have any $a, b, c, d \in \mathcal{V}$ with*



Proof: See, e.g., [Ret93, BdGR97, Gug07]. □

Let P be a term and let $\mathcal{W} \subseteq \mathcal{V}_P$. Then we can obtain from $\otimes P$ a new relation web $(\otimes P)|_{\mathcal{W}} = \langle \mathcal{W}; \mathcal{F}^\bullet, \mathcal{F}^\circ \rangle$ by simply removing all vertices not belonging to \mathcal{W} and all edges adjacent to them. Similarly we can obtain from P a term $P|_{\mathcal{W}}$ by removing in the term tree all leaves not in \mathcal{W} and then systematically removing all \circ - and \bullet -nodes that became unary by this. More formally, we define $a|_{\mathcal{W}} = a$ if $a \in \mathcal{W}$ and

$$[A \circ B]|_{\mathcal{W}} = \begin{cases} [A|_{\mathcal{W}} \circ B|_{\mathcal{W}}] & \text{if } \mathcal{V}_A \cap \mathcal{W} \neq \emptyset \text{ and } \mathcal{V}_B \cap \mathcal{W} \neq \emptyset \\ A|_{\mathcal{W}} & \text{if } \mathcal{V}_A \cap \mathcal{W} \neq \emptyset \text{ and } \mathcal{V}_B \cap \mathcal{W} = \emptyset \\ B|_{\mathcal{W}} & \text{if } \mathcal{V}_A \cap \mathcal{W} = \emptyset \text{ and } \mathcal{V}_B \cap \mathcal{W} \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

and similarly we define $(A \bullet B)|_{\mathcal{W}}$. Clearly we then have $\otimes(P|_{\mathcal{W}}) = (\otimes P)|_{\mathcal{W}}$, but note that $P|_{\mathcal{W}}$ is not necessarily a subterm of P . For example, let $P = [(a \bullet b) \circ (c \bullet [(d \bullet e) \circ f])]$ and $\mathcal{W} = \{a, c, f\}$. Then $P|_{\mathcal{W}} = [a \circ (c \bullet f)]$. If we have another term Q with $\mathcal{V}_P \cap \mathcal{V}_Q \neq \emptyset$ then we write $P|_Q$ to abbreviate $P|_{\mathcal{V}_P \cap \mathcal{V}_Q}$.

The term “relation web” first appears in [Gug07]. The basic idea, however, is much older. In graph theory, a graph $\langle \mathcal{V}; \mathcal{E}^\bullet \rangle$ not containing configuration (I0) is called P_4 -free. It is also called a *cograph* because its complement $\langle \mathcal{V}; \mathcal{E}^\circ \rangle$ has the same property. Cographs are used in [Ret96] to provide a correctness criterion for linear logic proof nets, where $\langle \bullet, \circ \rangle$ is $\langle \otimes, \wp \rangle$. One can also find the terms N -free or Z -free if configuration (I0) is forbidden, depending on how the picture is drawn. A comprehensive survey is for example [Möh89]. If \bullet is not commutative, but only associative, then \mathcal{E}^\bullet becomes a partial order, more precisely, a *series-parallel order* (by Proposition 4.2 it can be obtained from the singletons via series- and parallel composition of orders). The inclusion relation for these orders has been characterized by a rewriting system in [BdGR97].

4.3 Remark. Proposition 4.2 also scales to the case with more than two binary operations. For example in [Ret93, BdGR97, Gug07] it is proved for the case of two commutative operations and one non-commutative operation.

5 The Characterisation of Medial

For two terms P and Q , we write $P \blacktriangleleft Q$ if their relation webs obey the following three properties:

- (i) $\mathcal{V}_P = \mathcal{V}_Q$,
- (ii) $\mathcal{E}_P^\bullet \subseteq \mathcal{E}_Q^\bullet$ (or, equivalently, $\mathcal{E}_P^\circ \subseteq \mathcal{E}_Q^\circ$), and
- (iii) for all $a, d \in \mathcal{V}_P (= \mathcal{V}_Q)$ with $a \overset{\circ}{\underset{P}{\frown}} d$ and $a \overset{\bullet}{\underset{Q}{\frown}} d$, there are $b, c \in \mathcal{V}_P$ such that we have the following configurations

$$\text{in } \otimes P: \begin{array}{cc} a & \text{---} & b \\ & \diagdown & / \\ & c & \\ & / & \diagdown \\ c & \text{---} & d \end{array} \qquad \text{in } \otimes Q: \begin{array}{cc} a & \text{---} & b \\ & \diagup & \diagdown \\ & c & \\ & \diagdown & / \\ c & \text{---} & d \end{array} \tag{11}$$

The motivation for this definition is the following theorem.

5.1 Theorem. *For two terms P and Q we have $P \xrightarrow{*}_M Q$ iff $P \blacktriangleleft Q$.*

When proving this theorem, we make crucial use of two lemmas.

5.2 Lemma. *Let P and Q be terms with $P \xrightarrow{*}_M Q$. If P' is a subterm of P , then $P' \xrightarrow{*}_M Q|_{P'}$. And if Q_1 is a subterm of Q , then $P|_{Q_1} \xrightarrow{*}_M Q_1$.*

Proof: Since $P \xrightarrow{*}_M Q$, we have an $n \geq 0$ and terms R_0, \dots, R_n , such that $P \approx R_0 \xrightarrow{M} R_1 \xrightarrow{M} \dots \xrightarrow{M} R_n \approx Q$. We will say an R_i (for $0 \leq i \leq n$) is *nested* if there is a term $R \approx R_i$ which has a subterm $[(A_1 \bullet B_1) \circ (A_2 \bullet B_2)]$ such that $\mathcal{V}_{A_1} \cap \mathcal{V}_{P'} \neq \emptyset$ and $\mathcal{V}_{A_2} \cap \mathcal{V}_{P'} \neq \emptyset$ and $\mathcal{V}_{B_1} \cap \mathcal{V}_{P'} = \emptyset$. We first show that none of the R_i can be nested. Clearly $R_0 (\approx P)$ is not nested. Now we proceed by way of contradiction and pick the smallest i such that R_i is nested. Since R_i is obtained from R_{i-1} via a medial rewriting step, we can, without loss of generality, assume that $A_1 = [A \circ C]$ and $B_1 = [B \circ D]$ such that $\mathcal{V}_A \cap \mathcal{V}_{P'} \neq \emptyset$ and $\mathcal{V}_{[B \circ D]} \cap \mathcal{V}_{P'} = \emptyset$, and that R_{i-1} has $[(A \bullet B) \circ (C \bullet D) \circ (A_2 \bullet B_2)]$ as subterm. But then R_{i-1} is also nested. Contradiction. Now we define $R'_i = R_i|_{P'}$ for all $0 \leq i \leq n$. We are going to show that $R'_i \approx R'_{i+1}$ or $R'_i \xrightarrow{M} R'_{i+1}$ for all $0 \leq i \leq n$. We have $R_i \xrightarrow{M} R_{i+1}$. Hence, R_i has a subterm $[(A \bullet B) \circ (C \bullet D)]$ which is replaced by $([A \circ C] \bullet [B \circ D])$ in R_{i+1} . Now we proceed by way of contradiction: since $R'_i \not\approx R'_{i+1}$ we have (without loss of generality) that $\mathcal{V}_A \cap \mathcal{V}_{P'} \neq \emptyset$ and $\mathcal{V}_D \cap \mathcal{V}_{P'} \neq \emptyset$. Since additionally $R'_i \not\xrightarrow{M} R'_{i+1}$, we must have $\mathcal{V}_A \cap \mathcal{V}_{P'} = \emptyset$ or $\mathcal{V}_C \cap \mathcal{V}_{P'} = \emptyset$. Hence R_i is nested, which is a contradiction. Now the first statement of the lemma follows by an induction on n . The second statement is shown analogously. \square

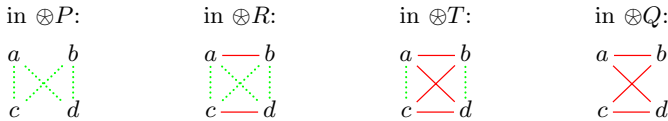
5.3 Lemma. *Let P and Q be terms with $P \blacktriangleleft Q$. If P' is a subterm of P , then $P' \blacktriangleleft Q|_{P'}$. And if Q_1 is a subterm of Q , then $P|_{Q_1} \blacktriangleleft Q_1$.*

Proof: For proving the first statement, let $Q' = Q|_{P'}$. We have $\mathcal{V}_{P'} = \mathcal{V}_{Q'}$ and $\mathcal{E}_P^\bullet \subseteq \mathcal{E}_{Q'}^\bullet$. Now let $a, d \in \mathcal{V}_{P'}$ with $a \overset{\circ}{\underset{P'}{\frown}} d$ and $a \overset{\bullet}{\underset{Q'}{\frown}} d$. Then we also have $a \overset{\circ}{\underset{Q'}{\frown}} d$

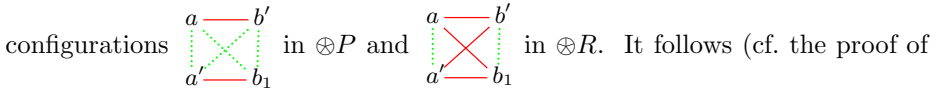
and $a \overset{\bullet}{\circ} d$, and therefore we have $b, c \in \mathcal{V}_P$ such that (III). In order to complete the proof of the lemma, we need to show that $b, c \in \mathcal{V}_{P'}$. By way of contradiction, assume that b occurs in the context of P' . Then b has the same first common ancestor with a and d in P . Hence, the edges (a, b) and (d, b) have the same color in $\otimes P$. Contradiction. The second statement is shown analogously. \square

5.4 Remark. It is important to observe that it is crucial for both lemmas that P' is a subterm of P (or that Q_1 is a subterm of Q). If we just have $P \blacktriangleleft Q$ (resp. $P \xrightarrow{*}_M Q$) and a subset $\mathcal{W} \subseteq \mathcal{V}_P$, then in general we do *not* have that $P|_{\mathcal{W}} \blacktriangleleft Q|_{\mathcal{W}}$ (resp. $P|_{\mathcal{W}} \xrightarrow{*}_M Q|_{\mathcal{W}}$). A simple example is given by $P = [(a \bullet b) \circ (c \bullet d)]$ and $Q = [(a \circ c) \bullet [b \circ d]]$ and $\mathcal{W} = \{a, b, d\}$. Then $P|_{\mathcal{W}} = [(a \bullet b) \circ d]$ and $Q|_{\mathcal{W}} = (a \bullet [b \circ d])$. We clearly have $P \blacktriangleleft Q$ (resp. $P \xrightarrow{*}_M Q$) but not $P|_{\mathcal{W}} \blacktriangleleft Q|_{\mathcal{W}}$ (resp. $P|_{\mathcal{W}} \xrightarrow{*}_M Q|_{\mathcal{W}}$).

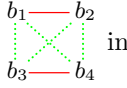
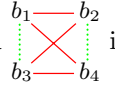
Proof of Theorem 5.1: First, assume we have $P \xrightarrow{*}_M Q$. Then there is an $n \geq 0$ with $P \xrightarrow{n}_M Q$. Obviously, we have $\mathcal{V}_P = \mathcal{V}_Q$ and $\mathcal{E}_P^{\bullet} \subseteq \mathcal{E}_Q^{\bullet}$. Hence Conditions (i) and (ii) are satisfied. For proving Condition (iii), we proceed by induction on n . For $n = 0$ this is trivial. Now let $n \geq 1$, and assume we have a and d with $a \overset{\circ}{\circ} d$ and $a \overset{\bullet}{\circ} d$. Then there are terms R and T such that $P \xrightarrow{*}_M R \xrightarrow{*}_M T \xrightarrow{*}_M Q$ and $a \overset{\circ}{\circ} d$ and $a \overset{\bullet}{\circ} d$. Because of Proposition 4.1, we can assume without loss of generality that that R has a subterm $[(A \bullet B) \circ (C \bullet D)]$, which is in T replaced by $[(A \circ C) \bullet [B \circ D]]$. We can without loss of generality assume that $a \in \mathcal{V}_A$ and $d \in \mathcal{V}_D$. Then we have for all $b \in \mathcal{V}_B$ and $c \in \mathcal{V}_C$ the following configurations:

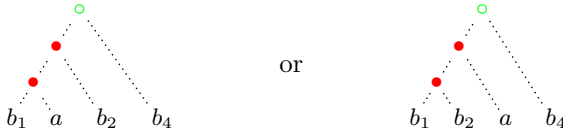


We will now show that there is a $b \in \mathcal{V}_B$ with $a \overset{\bullet}{\circ} b$ and $b \overset{\circ}{\circ} d$. For this, we need an auxiliary definition. For a term S and a constant $a \in \mathcal{V}_S$ we define a partial order $\overset{a}{\circ}_S$ on the set \mathcal{V}_S as follows: $b_1 \overset{a}{\circ}_S b_2$ iff the first common ancestor of a and b_2 in the term tree of S is also an ancestor of b_1 . For example, in (8) we have $b \overset{c}{\circ}_P e$, and d, e are incomparable wrt. $\overset{c}{\circ}_P$. Now pick $b_1 \in \mathcal{V}_B$ which is minimal wrt. $\overset{a}{\circ}_P$. We claim that $a \overset{\bullet}{\circ} b_1$. By way of contradiction, assume $a \overset{\circ}{\circ} b_1$. Then we apply the induction hypothesis to $P \xrightarrow{*}_M R$, which gives us a' and b' with



Lemma 5.3) that $b' \in \mathcal{V}_B$ and that $b' \overset{a}{\circ}_P b_1$, contradicting the minimality of b_1 . If $b_1 \overset{\circ}{\circ} d$, then we have found our desired b . So, assume $b_1 \overset{\bullet}{\circ} d$, and pick a $b_4 \in \mathcal{V}_B$ which is minimal wrt. $\overset{d}{\circ}_Q$. With a similar argument as above, we can show that

$b_4 \overset{\circ}{Q} d$. If $a \overset{\bullet}{P} b_4$, then, as before, we have our b . So, let us assume that $a \overset{\circ}{P} b_4$. Since we also have that $b_1 \overset{a}{P} b_4$ and $b_4 \overset{d}{Q} b_1$, it follows that $b_1 \overset{\circ}{P} b_4$ and $b_1 \overset{\bullet}{Q} b_4$. By Lemma 5.2 we have $P|_B \xrightarrow{*}_M B \xrightarrow{*}_M Q|_B$. Now we can apply the induction hypothesis to $P|_B \xrightarrow{*}_M Q|_B$ and get $b_2, b_3 \in \mathcal{V}_B$ such that we have  in $\otimes P|_B$ and  in $\otimes Q|_B$. Note that $b_2, b_3 \in \mathcal{V}_B$ and that $b_2 \overset{b_1}{P} b_4$. Hence, in the term tree for P , we have one of the following situations:



In both cases $a \overset{\bullet}{P} b_2$. Similarly, it follows that $b_2 \overset{\circ}{Q} d$. With a similar argumentation, we can find $c_2 \in \mathcal{V}_C$ with $c_2 \overset{\bullet}{P} d$ and $a \overset{\circ}{Q} c_2$. Hence, Condition (iii) is fulfilled, and we have $P \blacktriangleleft Q$.

Conversely, assume we have $P \blacktriangleleft Q$. We proceed by induction on the cardinality of \mathcal{V}_P , to show that $P \xrightarrow{*}_M Q$. The base case, where \mathcal{V}_P is a singleton, is trivial. Now we make a case analysis on the term structure of P and Q .

1. $P = [P' \circ P'']$ and $Q = [Q_1 \circ Q_2]$. We define the following four sets:

$$\mathcal{V}'_1 = \mathcal{V}_{P'} \cap \mathcal{V}_{Q_1}, \quad \mathcal{V}'_2 = \mathcal{V}_{P'} \cap \mathcal{V}_{Q_2}, \quad \mathcal{V}''_1 = \mathcal{V}_{P''} \cap \mathcal{V}_{Q_1}, \quad \mathcal{V}''_2 = \mathcal{V}_{P''} \cap \mathcal{V}_{Q_2}.$$

First, note that we cannot have that one of \mathcal{V}'_1 and \mathcal{V}''_2 is empty, and at the same time that one of \mathcal{V}'_2 and \mathcal{V}''_1 is empty because then one of $\mathcal{V}_{P'}, \mathcal{V}_{P''}, \mathcal{V}_{Q_1}, \mathcal{V}_{Q_2}$ would be empty, which is impossible. The remaining two possibilities of two empty sets are:

- If $\mathcal{V}'_2 = \emptyset$ and $\mathcal{V}''_1 = \emptyset$, then $\mathcal{V}_{P'} = \mathcal{V}_{Q_1}$ and $\mathcal{V}_{P''} = \mathcal{V}_{Q_2}$. Hence, by Lemma 5.3 we have $P' \blacktriangleleft Q_1$ and $P'' \blacktriangleleft Q_2$. By induction hypothesis we have therefore

$$P = [P' \circ P''] \xrightarrow{*}_M [Q_1 \circ P''] \xrightarrow{*}_M [Q_1 \circ Q_2] = Q$$

- If $\mathcal{V}'_1 = \emptyset$ and $\mathcal{V}''_2 = \emptyset$, then $\mathcal{V}_{P'} = \mathcal{V}_{Q_2}$ and $\mathcal{V}_{P''} = \mathcal{V}_{Q_1}$, and we proceed similarly.

Let us now assume that one of the four sets is empty, say $\mathcal{V}'_1 = \emptyset$. We let

$$P'_2 = P'|_{Q_2}, \quad P''_1 = P''|_{Q_1}, \quad P''_2 = P''|_{Q_2}.$$

Then $P'_2 = P'$ and $P''_1 \approx [P''_1 \circ P''_2]$ because $\mathcal{E}^{\circ}_Q \subseteq \mathcal{E}^{\circ}_P$. By Lemma 5.3 we have $P''_1 \blacktriangleleft Q_1$ and $[P'_2 \circ P''_2] \blacktriangleleft Q_2$. Hence, by induction hypothesis we have

$$P \approx [P'_2 \circ [P''_1 \circ P''_2]] \approx [P''_1 \circ [P'_2 \circ P''_2]] \xrightarrow{*}_M [Q_1 \circ [P'_2 \circ P''_2]] \xrightarrow{*}_M [Q_1 \circ Q_2] = Q$$

If one of $\mathcal{V}'_2, \mathcal{V}''_1, \mathcal{V}''_2$ is empty, we can proceed analogously. Let us now consider the case where none of $\mathcal{V}'_1, \mathcal{V}'_2, \mathcal{V}''_1, \mathcal{V}''_2$ is empty. Then we can define

$$P'_1 = P'|_{Q_1}, \quad P'_2 = P'|_{Q_2}, \quad P''_1 = P''|_{Q_1}, \quad P''_2 = P''|_{Q_2}.$$

We have $P' \approx [P'_1 \circ P'_2]$ and $P'' \approx [P''_1 \circ P''_2]$. By Lemma 5.3 we have $[P'_1 \circ P''_1] \blacktriangleleft Q_1$ and $[P'_2 \circ P''_2] \blacktriangleleft Q_2$. Hence, by induction hypothesis:

$$P \approx [[P'_1 \circ P'_2] \circ [P''_1 \circ P''_2]] \approx [[P'_1 \circ P''_1] \circ [P'_2 \circ P''_2]] \xrightarrow{*}_M [Q_1 \circ Q_2] = Q$$

2. $P = (P' \bullet P'')$ and $Q = (Q_1 \bullet Q_2)$. This is analogous to the previous case.
3. $P = (P' \bullet P'')$ and $Q = [Q_1 \circ Q_2]$. As before, we let

$$\mathcal{V}'_1 = \mathcal{V}_{P'} \cap \mathcal{V}_{Q_1}, \mathcal{V}'_2 = \mathcal{V}_{P'} \cap \mathcal{V}_{Q_2}, \mathcal{V}''_1 = \mathcal{V}_{P''} \cap \mathcal{V}_{Q_1}, \mathcal{V}''_2 = \mathcal{V}_{P''} \cap \mathcal{V}_{Q_2}.$$

Note that if $\mathcal{V}'_1 \neq \emptyset$ and $\mathcal{V}''_2 \neq \emptyset$ then we have immediately a contradiction to Condition (ii), and similarly if $\mathcal{V}'_2 \neq \emptyset$ and $\mathcal{V}''_1 \neq \emptyset$. Hence, one of \mathcal{V}'_1 and \mathcal{V}''_2 must be empty, and one of \mathcal{V}'_2 and \mathcal{V}''_1 must be empty. But this is impossible as observed in Case 1 above.

4. $P = [P' \circ P'']$ and $Q = (Q_1 \bullet Q_2)$. This is the most interesting case. As before, we let

$$\mathcal{V}'_1 = \mathcal{V}_{P'} \cap \mathcal{V}_{Q_1}, \mathcal{V}'_2 = \mathcal{V}_{P'} \cap \mathcal{V}_{Q_2}, \mathcal{V}''_1 = \mathcal{V}_{P''} \cap \mathcal{V}_{Q_1}, \mathcal{V}''_2 = \mathcal{V}_{P''} \cap \mathcal{V}_{Q_2}.$$

We first show that none of the sets $\mathcal{V}'_1, \mathcal{V}''_1, \mathcal{V}'_2, \mathcal{V}''_2$ is empty. So, assume by way of contradiction, that $\mathcal{V}''_1 = \emptyset$. By a similar argumentation as before it follows that $\mathcal{V}'_1 \neq \emptyset$ and $\mathcal{V}''_2 \neq \emptyset$. So, pick $a \in \mathcal{V}'_1$ and $d \in \mathcal{V}''_2$. We have $a \overset{\circ}{P} d$ and $a \overset{\bullet}{Q} d$. Since $P \blacktriangleleft Q$, we have $b, c \in \mathcal{V}_P$ such that (III). Because $c \overset{\bullet}{P} d$ we must have that $c \in \mathcal{V}_{P''}$, and because of $a \overset{\circ}{Q} c$, we must have that $c \in \mathcal{V}_{Q_1}$. Hence $c \in \mathcal{V}''_1$. Contradiction. We can therefore define:

$$P'_1 = P'|_{Q_1}, \quad P'_2 = P'|_{Q_2}, \quad P''_1 = P''|_{Q_1}, \quad P''_2 = P''|_{Q_2},$$


and



$$Q'_1 = Q_1|_{P'}, \quad Q'_2 = Q_2|_{P'}, \quad Q''_1 = Q_1|_{P''}, \quad Q''_2 = Q_2|_{P''}.$$

We now want to show that $P'_1 \blacktriangleleft Q'_1$. But by Remark 5.4 we cannot apply Lemma 5.3. However, we have $\mathcal{V}_{P'_1} = \mathcal{V}_{Q'_1}$ and $\mathcal{E}_{P'_1} \subseteq \mathcal{E}_{Q'_1}$. Now let $a, d \in \mathcal{V}_{P'_1}$ with $a \overset{\circ}{P'_1} d$ and $a \overset{\bullet}{Q'_1} d$. Hence, we have $a \overset{\circ}{P} d$ and $a \overset{\bullet}{Q} d$. Since $P \blacktriangleleft Q$, we have $b, c \in \mathcal{V}_P$ such that (III). Note that because $a, d \in \mathcal{V}_{P'}$, we also have $b \in \mathcal{V}_{P'}$ (otherwise we would have $a \overset{\circ}{P} b$) and $c \in \mathcal{V}_{P'}$ (otherwise we would have $c \overset{\circ}{P} d$). Similarly, because $a, d \in \mathcal{V}_{Q_1}$, we also have $b, c \in \mathcal{V}_{Q_1}$ (otherwise we would have $a \overset{\bullet}{Q} c$ and $b \overset{\bullet}{Q} d$, respectively). Hence $b, c \in \mathcal{V}_{P'_1}$, and therefore $P'_1 \blacktriangleleft Q'_1$. Similarly, we get $P''_1 \blacktriangleleft Q''_1$ and $P'_2 \blacktriangleleft Q'_2$ and $P''_2 \blacktriangleleft Q''_2$. Hence, we have by induction hypothesis

$$P'_1 \xrightarrow{*}_M Q'_1, \quad P''_1 \xrightarrow{*}_M Q''_1, \quad P'_2 \xrightarrow{*}_M Q'_2, \quad P''_2 \xrightarrow{*}_M Q''_2. \quad (12)$$

Now let $P'_{12} = (P'_1 \bullet P'_2)$. We clearly have $\mathcal{V}_{P'} = \mathcal{V}_{P'_{12}}$ and $\mathcal{E}_{P'} \subseteq \mathcal{E}_{P'_{12}}$. Now let us assume we have $a, d \in \mathcal{V}_{P'}$ with $a \overset{\circ}{P'} d$ and $a \overset{\bullet}{P'_{12}} d$. Then we must have $a \in \mathcal{V}_{P'_1}$ and $d \in \mathcal{V}_{P'_2}$, or vice versa (otherwise the two edges would have the same color in P' and P'_{12}). Hence, we have $a \overset{\circ}{P} d$ and $a \overset{\bullet}{Q} d$. Since $P \blacktriangleleft Q$, we have $b, c \in \mathcal{V}_Q$ such that (III). Note that because $a, d \in \mathcal{V}_{P'}$, we also have $b, c \in \mathcal{V}_{P'}$ (otherwise we would have $a \overset{\circ}{P} b$ and $d \overset{\circ}{P} c$). This means we have in

$\otimes P'$ the configuration . Since we have $a \overset{\circ}{Q} c$ and $b \overset{\circ}{Q} d$, we must also have $a \overset{\circ}{P'} c$ and $b \overset{\circ}{P'} d$. And since we have $a \overset{\bullet}{P} b$ and $c \overset{\bullet}{P} d$, we also have $a \overset{\bullet}{P'} b$ and $c \overset{\bullet}{P'} d$. Furthermore, we have $a \overset{\bullet}{P'} d$ (because $a \in \mathcal{V}_{P'}$ and $d \in \mathcal{V}_{P'}$).

Hence, we have in $\otimes P'_{12}$ the configuration . By Proposition 4.2, we must have . Hence, $P' \blacktriangleleft (P'_1 \bullet P'_2)$. By the same argumentation, we get $P'' \blacktriangleleft (P''_1 \bullet P''_2)$ and $[Q'_1 \circ Q''_1] \blacktriangleleft Q_1$ and $[Q'_2 \circ Q''_2] \blacktriangleleft Q_2$. By induction hypothesis we have therefore

$$\begin{aligned} P' &\xrightarrow[M]{*} (P'_1 \bullet P'_2) & [Q'_1 \circ Q''_1] &\xrightarrow[M]{*} Q_1 \\ P'' &\xrightarrow[M]{*} (P''_1 \bullet P''_2) & [Q'_2 \circ Q''_2] &\xrightarrow[M]{*} Q_2 \end{aligned} \tag{13}$$

Now we can combine (12) and (13) to get

$$\begin{aligned} [P' \circ P''] &\xrightarrow[M]{*} [(P'_1 \bullet P'_2) \circ (P''_1 \bullet P''_2)] \xrightarrow[M]{*} ((P'_1 \circ P''_1) \bullet [P'_2 \circ P''_2]) \\ &\xrightarrow[M]{*} ([Q'_1 \circ Q''_1] \bullet [Q'_2 \circ Q''_2]) \xrightarrow[M]{*} (Q_1 \bullet Q_2) \end{aligned}$$

In other words: $P \xrightarrow[M]{*} Q$. □

5.5 Corollary. *The relation $\blacktriangleleft \subseteq \mathcal{T} \times \mathcal{T}$ is transitive.*

6 Related Results

Let us compare our result to the one in [BdGR97], where one of the two binary operations was not commutative but only associative. Although this has some consequences for the characterization of relation webs (Proposition 4.2), the consequences for the main result (Theorem 5.1) are only cosmetic. For this reason let us recall here the commutative version of the results in [BdGR97]. Let P be the rewriting system

$$\begin{aligned} ([x \circ y] \bullet [w \circ z]) &\rightarrow [(x \bullet w) \circ (y \bullet z)] \\ (x \bullet [y \circ z]) &\rightarrow [(x \bullet y) \circ z] \\ (x \bullet y) &\rightarrow [x \circ y] \end{aligned} \tag{14}$$

Note that it is *not* a typo that the first rewrite rule is the inversion of medial. Analogous to $\xrightarrow[M]{*}$, we define $\xrightarrow[P]{*}$ to be the transitive closure of the rewriting relation via (14) modulo AC. The result of [BdGR97] can be stated as follows:

6.1 Theorem. *For terms P, Q we have $P \xrightarrow[P]{*} Q$ iff $\mathcal{V}_P = \mathcal{V}_Q$ and $\mathcal{E}_P^\circ \subseteq \mathcal{E}_Q^\circ$.*

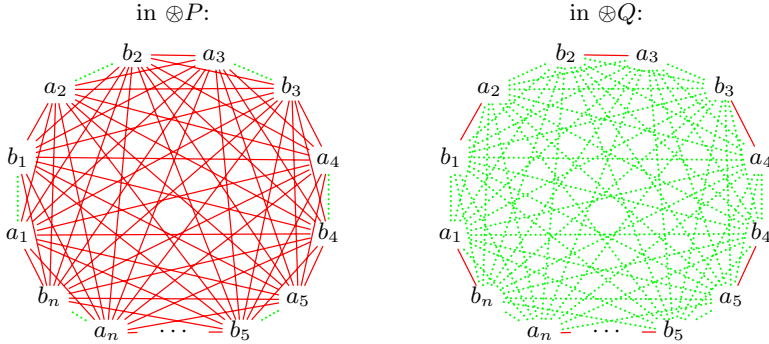
In other words, the main difference to Theorem 5.1 is that the Condition (iii) is absent in [BdGR97]. Let us now look at the case where we remove the first rule from P . Let S be the rewrite system

$$\begin{aligned} (x \bullet [y \circ z]) &\rightarrow [(x \bullet y) \circ z] \\ (x \bullet y) &\rightarrow [x \circ y] \end{aligned} \tag{15}$$

We define $P \xrightarrow{*}_S Q$ as above. The characterization of this relation is the following:

6.2 Theorem. *We have $P \xrightarrow{*}_S Q$ if and only if $\mathcal{V}_P = \mathcal{V}_Q$, and for all $n \geq 1$ and all subsets $\mathcal{W} = \{a_1, b_1, \dots, a_n, b_n\} \subseteq \mathcal{V}_P$ we do not have that $P|_{\mathcal{W}} \approx ([a_1 \circ b_1] \bullet \dots \bullet [a_n \circ b_n])$ and $Q|_{\mathcal{W}} \approx [(b_1 \bullet a_2) \circ (b_2 \bullet a_3) \circ \dots \circ (b_n \bullet a_1)]$*

In other words, we are not allowed to have the following configurations in the relation webs of P and Q :



Note that $\mathcal{E}_P^\circ \subseteq \mathcal{E}_Q^\circ$ follows by letting $n = 1$.

6.3 Remark. This characterization is simply an alternative formulation of the correctness criterion for proof nets for multiplicative linear logic [Ret96]. For this, we have to read the \bullet as *tensor* \otimes , and the \circ as *par* \wp . Then, the rule

$$(x \otimes [y \wp z]) \rightarrow [(x \otimes y) \wp z]$$

is also called *switch* [Gug07], *weak distributivity* [BCST96], or *dissociativity* [DP04]. The rule

$$(x \otimes y) \rightarrow [x \wp y]$$

is called *mix*. The condition in Theorem 6.2 is equivalent to the acyclicity condition in proof nets [DR89, 2]

It is interesting to note the different nature of the three characterizations of the rewrite systems M, P, and S. This is the reason for the difficulty to give a characterization of the rewrite system MS, which combines M and S:

$$\begin{aligned} (x \bullet y) \circ (w \bullet z) &\rightarrow [(x \circ w) \bullet [y \circ z]] \\ (x \bullet [y \circ z]) &\rightarrow [(x \bullet y) \circ z] \\ (x \bullet y) &\rightarrow [x \circ y] \end{aligned} \tag{16}$$

² If mix is absent, then an additional condition (connectedness) would be needed. For more details on the relation between S and linear logic, see, e.g., [DHPP99, Ret93, Gug07, Str03a], and for relating the condition in Theorem 6.2 to multiplicative proof nets, see, e.g., [Ret03]. For more information on mix, see [FR94], and for a direct proof of Theorem 6.2, see, e.g., [Str03b, Str03a].

6.4 Open Problem: Find a characterization of the rewrite relation $\xrightarrow[\text{MS}]{*}$ in terms of relation webs.

7 Application in Proof Theory

The motivation for stating the open problem concluding the previous section is the increasing importance of the relation $\xrightarrow[\text{MS}]{*}$ for the proof theory of classical propositional logic [BT01, Lam06, Str05]. To see this, we have to read the \bullet as *conjunction* \wedge and the \circ as *disjunction* \vee .

A central ingredient to logic is the notion of duality. For dealing with this, we let the set of constant symbols come in pairs: for every a there is its dual \bar{a} . Then the terms are the formulas in negation normal form and the constants are the literals. If a formula I is of the shape

$$([a_1 \circ \bar{a}_1] \bullet [a_2 \circ \bar{a}_2] \bullet \cdots \bullet [a_n \circ \bar{a}_n])$$

for some $n \geq 1$ and constants a_1, a_2, \dots, a_n , then we say I is an *initial formula*.

It is well-known that classical logic is multiplicative linear logic plus contraction and weakening. Let us therefore introduce two more rewrite systems. Let W be the rewrite system containing only the rule

$$x \rightarrow [x \circ y] \tag{17}$$

and let C be the system containing only the rule

$$[x \circ x] \rightarrow x \tag{18}$$

Now let $K1 = S \cup W \cup C$. Then we have the following theorem, which says that a proof in classical logic is a rewrite path in $K1$.

7.1 Theorem. *A formula Q is a Boolean tautology if and only if there is an initial formula I with $I \xrightarrow[\text{K1}]{*} Q$.* [BT01]

As already mentioned in Section 2, we can with medial reduce contraction to literals. Let C' be the rewrite system consisting of a rule

$$[a \circ a] \rightarrow a \tag{19}$$

for every constant symbol (including their duals). If we let $K2 = MS \cup W \cup C'$, then we have

7.2 Theorem. *Let P and Q be formulas. Then $P \xrightarrow[\text{K1}]{*} Q$ iff $P \xrightarrow[\text{K2}]{*} Q$.* [BT01]

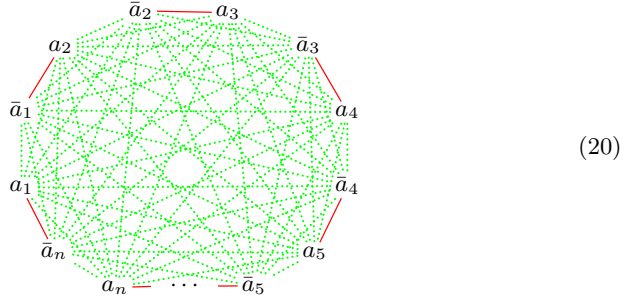
While [BT01] and related work (e.g., [GS01, Gug07, Bri03, Str03a]) are mainly concerned with the syntactic manipulation of terms/formulas, Hughes proposes in [Hug06] the notion of *combinatorial proof*, which is based on a variant of Theorem 6.2 and the notion of *skew fibration*: Given two prewebs $\mathcal{G}_1 = \langle \mathcal{V}_1; \mathcal{E}_1^\bullet, \mathcal{E}_1^\circ \rangle$ and $\mathcal{G}_2 = \langle \mathcal{V}_2; \mathcal{E}_2^\bullet, \mathcal{E}_2^\circ \rangle$, then a skew fibration $h: \mathcal{G}_1 \rightarrow \mathcal{G}_2$ is a mapping $\mathcal{V}_1 \rightarrow \mathcal{V}_2$ such that

- (a) $(a, b) \in \mathcal{E}_1^\bullet$ implies $(h(a), h(b)) \in \mathcal{E}_2^\bullet$ (i.e., h is a graph homomorphism for the red edges), and

(b) for all $a \in \mathcal{V}_1$ and $d \in \mathcal{V}_2$, if $(h(a), d) \in \mathcal{E}_2^\bullet$, then there is a $b \in \mathcal{V}_1$ with $(a, b) \in \mathcal{E}_1^\bullet$ and $(h(b), d) \notin \mathcal{E}_2^\bullet$.

A combinatorial proof of a Boolean formula Q is a skew fibration $h: \otimes P \rightarrow \otimes Q$ for a formula P such that

(c) $\otimes P$ does not contain a configuration



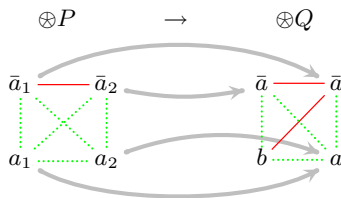
for any $n \geq 1$ and constants a_1, a_2, \dots, a_n , and

(d) h maps only non-negated constants to non-negated constants and negated constants to negated ones.

7.3 Theorem. A formula Q is a Boolean tautology, if and only if it has a combinatorial proof. Hug06

7.4 Remark. Note that for Theorems 7.1 and 7.3 to make sense, we have to allow more than one occurrence of a constant in a formula. This means that in the relation web $\otimes P$ of a formula P , the set \mathcal{V}_P is the set of constant occurrences. Then we can call a map $h: \mathcal{V}_P \rightarrow \mathcal{V}_Q$ label preserving if the name of a constant is not changed by h .

To give an example, we show here the combinatorial proof of Pierce’s law $Q = [([\bar{a} \vee b] \wedge \bar{a}) \vee a]$, taken from Hug06. We let $P = [([\bar{a}_1 \wedge \bar{a}_2] \vee a_1 \vee a_2)]$. The skew fibration $h: \otimes P \rightarrow \otimes Q$ is given as follows:



7.5 Theorem. Let P and Q be formulas. Then $P \blacktriangleleft Q$ if and only if $\mathcal{V}_P = \mathcal{V}_Q$ and the identity function on \mathcal{V}_P is a skew fibration $\otimes P \rightarrow \otimes Q$.

Proof: First, assume $P \blacktriangleleft Q$. Since $\mathcal{E}_P^\bullet \subseteq \mathcal{E}_Q^\bullet$, Condition (a) above is fulfilled. Now let $a, d \in \mathcal{V}_P$ with $a \overset{\bullet}{\underset{Q}{\blacktriangleright}} d$. If $a \overset{\bullet}{\underset{P}{\blacktriangleright}} d$, then we let $b = d$ and we are done. If $a \overset{\circ}{\underset{P}{\blacktriangleright}} d$, then we have $b, c \in \mathcal{V}_P$ with (11). Now b has the desired properties. Conversely, assume that $\mathcal{V}_P = \mathcal{V}_Q$ and the identity $\mathcal{V}_P \rightarrow \mathcal{V}_Q$ is a skew fibration.

By (a) we have $\mathcal{E}_P^\bullet \subseteq \mathcal{E}_Q^\bullet$. Now let $a, d \in \mathcal{V}_P$ with $a \overset{\circ}{\underset{P}{\frown}} d$ and $a \overset{\bullet}{\underset{Q}{\frown}} d$. Then by (b) there is a $b \in \mathcal{V}_P$ with $a \overset{\bullet}{\underset{P}{\frown}} b$ and $b \overset{\circ}{\underset{Q}{\frown}} d$. Since $\mathcal{E}_P^\bullet \subseteq \mathcal{E}_Q^\bullet$, we also have $a \overset{\bullet}{\underset{Q}{\frown}} b$ and $b \overset{\circ}{\underset{P}{\frown}} d$. By exchanging the roles of a and d and applying (b) again, we get $c \in \mathcal{V}_P$ with $d \overset{\bullet}{\underset{P}{\frown}} c$ and $c \overset{\circ}{\underset{Q}{\frown}} a$. Since $\mathcal{E}_P^\bullet \subseteq \mathcal{E}_Q^\bullet$, it follows that $d \overset{\bullet}{\underset{Q}{\frown}} c$ and $c \overset{\circ}{\underset{P}{\frown}} a$. Hence $c \neq b$. By Proposition 4.2, we conclude that $b \overset{\circ}{\underset{P}{\frown}} c$ and $b \overset{\bullet}{\underset{Q}{\frown}} c$. \square

In the following, we establish a precise relation between the notion of proof as rewriting path (in a deep inference deductive system) and the notion of proof as a combinatorial object using relation webs and skew fibrations. For this, we first have to characterize the rewrite systems W and C' . Let P and Q be formulas. A map $w: \otimes P \rightarrow \otimes Q$ is called a *weakening*, if

- (e) w is an injective skew fibration, and
- (f) for all $a, b \in \mathcal{V}_P$, we have $a \overset{\bullet}{\underset{P}{\frown}} b$ iff $w(a) \overset{\bullet}{\underset{Q}{\frown}} w(b)$.

A map $c: \otimes P \rightarrow \otimes Q$ is called an *atomic contraction*, if

- (g) c is surjective, and
- (h) for all $a, b \in \mathcal{V}_P$, we have $a \overset{\bullet}{\underset{P}{\frown}} b$ iff $c(a) \overset{\bullet}{\underset{Q}{\frown}} c(b)$.

Note that it follows that c is a skew fibration. We have the following:

7.6 Proposition. *For all formulas P and Q ,*

1. $P \xrightarrow[W]{*} Q$ iff there is a label preserving weakening $w: \otimes P \rightarrow \otimes Q$.
2. $P \xrightarrow[C']{*} Q$ iff there is a label preserving atomic contraction $c: \otimes P \rightarrow \otimes Q$.

Proof: The “only if” direction is trivial for both statements. The “if” direction for the first statement follows by observing that condition (b) implies that for all d not in the image of w there is in Q a subformula D containing only material (including d) not appearing in P , and a subformula B containing only material (including b) appearing in P , such that $[B \circ D]$ is also a subformula of Q . Injectivity and Condition (f) ensure that B is also a subformula of P . Hence, we can rewrite B into $[B \circ D]$. For the second statement it suffices to note that whenever two occurrences of a constant a in P are mapped onto the same occurrence in Q , then they must appear as subformula $[a \circ a]$ in P . \square

7.7 Lemma. *A label preserving skew fibration $h: \mathcal{V}_P \rightarrow \mathcal{V}_Q$ is surjective if and only if there is a formula R with $\mathcal{V}_R = \mathcal{V}_P$ such that $P \blacktriangleleft R$ and h is an atomic contraction when seen as map $\otimes R \rightarrow \otimes Q$.*

Proof: Let h be surjective. We construct R from Q by replacing each constant occurrence a by $[a \circ \dots \circ a]$ where the number of a ’s is the cardinality of the preimage $h^{-1}(a)$ in P . Then obviously the canonical map $\mathcal{V}_R \rightarrow \mathcal{V}_Q$ is an atomic contraction, and the identity map $\mathcal{V}_P \rightarrow \mathcal{V}_R$ inherits from h the property of being a skew fibration. Finally we apply Theorem 7.5. The converse follows from the fact that the composition of a skew fibration with an atomic contraction is again a skew fibration³. \square

³ An anonymous referee pointed out that it is in general not true that the composition of two skew fibrations is again a skew fibration because they are defined on prewebs.

Now we can put everything together to give a combinatorial proof for the following theorem:

7.8 Theorem. *A formula Q is a Boolean tautology if and only if there is an initial formula I , such that $I \xrightarrow{*}_S P \xrightarrow{*}_M R \xrightarrow{*}_{C'} S \xrightarrow{*}_W Q$ for some formulas P , R , and S .*

Proof: The “if” direction follows immediately from Theorems 7.1 and 7.2. For the “only if” direction we start with the combinatorial proof for Q given by Theorem 7.3. We have a skew fibration $h: \oplus P \rightarrow \oplus Q$. By Theorem 6.2 and Condition (c) we can obtain an initial formula I with $I \xrightarrow{*}_S P$. Now we let $\mathcal{V}_S \subseteq \mathcal{V}_Q$ be the image of $h: \mathcal{V}_P \rightarrow \mathcal{V}_Q$, and let $S = Q|_{\mathcal{V}_S}$. This gives us a surjective skew fibration $h': \oplus P \rightarrow \oplus S$. We can rename in P (and in I) all appearing constants such that h' becomes label preserving. Then we apply Lemma 7.7 to get R . By Theorem 5.1 we have $P \xrightarrow{*}_M R$, and by Proposition 7.6.2 we have $R \xrightarrow{*}_{C'} S$. Finally, note that the embedding $\oplus S \rightarrow \oplus Q$ is a weakening. So, by Proposition 7.6.1 we get $S \xrightarrow{*}_W Q$. \square

7.9 Remark. The proof of Theorem 7.8, together with the rule permutation results in the calculus of structures [Brü03] can be used to show that skew fibrations are closed under composition when their definition is restricted to relation webs (cf. Footnote 3).

8 Conclusions and Future Work

We have shown a combinatorial criterion for characterizing rewriting via medial modulo associativity and commutativity. This has been used for giving a combinatorial proof to a proof theoretic statement. So far, statements as in Theorem 7.8, also called *decomposition theorems* [Str03b, Brü03], have been proved via tedious permutations of inference rules in the calculus of structures. An interesting question for future research is whether these proofs can be simplified in general via a combinatorial analysis as carried out in this paper.

A second line of research is in the area of coherence problems in category theory. There, the question is not the existence of rewriting paths, but the identity of rewriting paths. Some investigation in this direction for rewriting via M can be found in [DP07]. For the system MS , see [Lam06], and for all of $K1$ and/or $K2$, see [Str05].

References

[BCST96] Blute, R., Cockett, R., Seely, R., Trimble, T.: Natural deduction and coherence for weakly distributive categories. *Journal of Pure. and Applied Algebra* 113, 229–296 (1996)

[BN98] Baader, F., Nipkow, T.: *Term Rewriting and All That*. Cambridge University Press, Cambridge (1998)

- [BdGR97] Bechet, D., de Groote, P., Retoré, C.: A complete axiomatisation of the inclusion of series-parallel partial orders. In: Comon, H. (ed.) *Rewriting Techniques and Applications*. LNCS, vol. 1232, pp. 230–240. Springer, Heidelberg (1997)
- [Brü03] Brünnler, K.: *Deep Inference and Symmetry for Classical Proofs*. PhD thesis, Technische Universität Dresden (2003)
- [BT01] Brünnler, K., Fernanto Tiu, A.: A local system for classical logic. In: Nieuwenhuis, R., Voronkov, A. (eds.) *LPAR 2001*. LNCS (LNAI), vol. 2250, pp. 347–361. Springer, Heidelberg (2001)
- [DG04] Di Gianantonio, P.: Structures for multiplicative cyclic linear logic: Deepness vs cyclicity. In: Marcinkowski, J., Tarlecki, A. (eds.) *CSL 2004*. LNCS, vol. 3210, pp. 130–144. Springer, Heidelberg (2004)
- [DHPP99] Devarajan, H., Hughes, D., Plotkin, G., Pratt, V.R.: Full completeness of the multiplicative linear logic of Chu spaces. In: *14th IEEE Symposium on Logic in Computer Science (LICS 1999)* (1999)
- [DP04] Došen, K., Petrić, Z.: *Proof-Theoretical Coherence*. KCL Publications, London (2004)
- [DP07] Došen, K., Petrić, Z.: *Intermutation*. Mathematical Institute, Belgrade (2007)
- [DR89] Danos, V., Regnier, L.: The structure of multiplicatives. *Annals of Mathematical Logic* 28, 181–203 (1989)
- [FR94] Fleury, A., Retoré, C.: The mix rule. *Mathematical Structures in Computer Science* 4(2), 273–285 (1994)
- [GS01] Guglielmi, A., Straßburger, L.: Non-commutativity and MELL in the calculus of structures. In: Fribourg, L. (ed.) *CSL 2001 and EACSL 2001*. LNCS, vol. 2142, pp. 54–68. Springer, Heidelberg (2001)
- [Gug07] Guglielmi, A.: A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1) (2007)
- [Hug06] Hughes, D.J.D.: Proofs Without Syntax. *Annals of Mathematics* 164(3), 1065–1076 (2006)
- [Lam06] Lamarche, F.: Exploring the gap between linear and classical logic, (Accepted for publication in TAC) (2006)
- [Mac71] Mac lane, S.: *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag (1971)
- [Möh89] Möhring, R.H.: Computationally tractable classes of ordered sets. In: Rival, I. (ed.) *Algorithms and Order*, pp. 105–194. Kluwer Acad. Publ, Boston, MA (1989)
- [Ret93] Retoré, C.: *Réseaux et Séquents Ordonnés*. Thèse de Doctorat, spécialité mathématiques, Université Paris VII (February 1993)
- [Ret96] Retoré, C.: Perfect matchings and series-parallel graphs: multiplicatives proof nets as R&B-graphs. *ENTCS*,3 (1996)
- [Ret03] Retoré, C.: Handsome proof-nets: perfect matchings and cographs. *Theoretical Computer Science* 294(3), 473–488 (2003)
- [Str02] Straßburger, L.: A local system for linear logic. In: Baaz, M., Voronkov, A. (eds.) *LPAR 2002*. LNCS (LNAI), vol. 2514, pp. 388–402. Springer, Heidelberg (2002)
- [Str03a] Straßburger, L.: *Linear Logic and Noncommutativity in the Calculus of Structures*. PhD thesis, Technische Universität Dresden (2003)
- [Str03b] Straßburger, L.: MELL in the Calculus of Structures. *Theoretical Computer Science* 309(1–3), 213–285 (2003)
- [Str05] Straßburger, L.: On the axiomatisation of Boolean categories with and without medial Accepted for publication in TAC (2005)

The Maximum Length of Mu-Reduction in Lambda Mu-Calculus

Makoto Tatsuta

National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
tatsuta@nii.ac.jp

Abstract. This paper gives the exact number of the maximum length of mu-reduction and permutative conversions for an untyped term in lambda mu-calculus with disjunction. This number is described by using induction on the number of symbols in a term. It is also shown that left-most short reduction and innermost null reduction produce the longest reduction sequence.

1 Introduction

$\lambda\mu$ -calculus given in [9] has been intensively studied, for example [1,2,3,6,7,8], because it is important in both mathematical logic and computer science; this system corresponds to classical natural deduction by Curry-Howard isomorphism, and this system gives us a theory of functional programming languages with continuation mechanism.

Recently permutative conversions have been studied actively [1,3,4,5,6,8,11]. Permutative conversions transform a proof with a disjunction or existential quantification elimination rule followed by an elimination rule into a proof with the second rule in the minor deduction of the first rule. Permutative conversions are indispensable for normalizing a proof in a natural deduction system with disjunction, since without permutative conversions, a normal proof fails to have the subformula property. Permutative conversions also give program transformation for if-then-else statements.

One of nice properties for typed $\lambda\mu$ -calculus is strong normalization [9]. So far there are two techniques to prove it; one is saturated-set semantics and the other is CPS translation. The CPS translation reduces strong normalization of typed $\lambda\mu$ -calculus to that of typed λ -calculus [3,7]. In that proof, strong normalization of μ -reduction has to be proved separately. The strong normalization of untyped μ -reduction is proved in [3] by giving an upper bound of reduction steps. The strong normalization of untyped symmetric μ -reduction is proved in [2] by induction on a term.

This paper gives the exact number of the maximum length of μ -reduction in untyped $\lambda\mu$ -calculus with disjunction. This number for a term is expressed by induction on the number of symbols in the term. So far the exact counting of the maximum μ -reduction steps has not been studied. This paper also analyzes the maximum number of steps of μ -reduction with permutative conversions.

This problem is itself interesting in the view point of combinatorics. Moreover these results provide us perspective understanding of μ -reduction and permutative conversions. We will also give reduction strategy that produces the longest reduction sequence.

To count the exact maximum steps, we will use the property that the maximum steps for each redex can be calculated independently of the other redexes, and the maximum steps for a term can be obtained as the sum of the maximum steps for all the redexes. To implement this idea, we will use a context-argument reduction tree. The number of inner nodes in this tree will give the maximum number of steps for a specific redex, and the number of leaves in this tree will give the maximum number of copies by the specific redex. Moreover this tree is an invariant under effective non-context-argument reduction. According to this counting, we will also give two reduction strategies, leftmost short reduction and innermost null reduction, and prove that these strategies actually give the longest reduction sequence.

Section 2 defines the system $\lambda\mu^\vee$, $\lambda\mu$ -calculus with disjunction. The maximum length of reduction steps is described in Section 3. Section 4 defines leftmost short reduction and innermost null reduction, and gives intuitive explanation for our description of the maximum length. Section 5 proves that it is actually the maximum length, and also proves that those reduction strategies produce the longest reduction sequence. Section 6 gives concluding remarks.

2 The Lambda Mu-Calculus with Disjunction

We will give the definition of the system $\lambda\mu^\vee$, the untyped $\lambda\mu$ -calculus with disjunction. This system is obtained from $\lambda\mu$ -calculus in [9] by adding disjunction in natural deduction given in [10]. Its reduction system has permutative conversions for disjunction.

Definition 2.1 (Language)

Variables x, y, z, u, v, w, \dots

Names α, β, \dots

Terms $M, N, L, P, Q, R, S ::= x | MN | [\alpha]M | \mu\alpha.M | M[x.N, y.L]$

MN is an application, $[\alpha]M$ is a named term, and $\mu\alpha.M$ is a μ -abstraction term. $M[x.N, y.L]$ is a disjunction elimination term that comes from the disjunction elimination inference rule

$$\frac{\begin{array}{ccc} [x : A] & [y : B] & \\ \vdots & \vdots & \\ M : A \vee B & N : C & L : C \end{array}}{M[x.N, y.L] : C}$$

in natural deduction [10]. The variable x in N and the variable y in L are bound in $M[x.N, y.L]$.

Notation. The symbol $=$ is used for the syntactical identity modulo bound variable renaming. The set $\text{FN}(M)$ of free names in the term M is defined in

a standard way by: $\text{FN}(x) = \phi$, $\text{FN}(MN) = \text{FN}(M) \cup \text{FN}(N)$, $\text{FN}([\alpha]M) = \{\alpha\} \cup \text{FN}(M)$, $\text{FN}(\mu\alpha.M) = \text{FN}(M) - \{\alpha\}$, and $\text{FN}(M[x.N, y.L]) = \text{FN}(M) \cup \text{FN}(N) \cup \text{FN}(L)$.

Remark. The $\lambda\mu$ -calculus with disjunction usually has the following term syntax: $M ::= x|\lambda x.M|MN|[\alpha]M|\mu\alpha.M|\langle 0, M \rangle|\langle 1, M \rangle|M[x.M, x.M]$. However, since we will only discuss μ -reduction and permutative conversions, for simplicity we excluded lambda abstraction $\lambda x.M$ and injections $\langle 0, M \rangle$ and $\langle 1, M \rangle$ for disjunction in our term syntax.

Substitution $M[x := N]$ is defined in a familiar way.

We define eliminators by $E, F ::= M|[x.M, y.N]$.

Structural substitution $M[\alpha^* := E]$ is defined in a standard way as follows:

$$\begin{aligned} x[\alpha^* := E] &= x \\ (ML)[\alpha^* := E] &= (M[\alpha^* := E])(L[\alpha^* := E]) \\ ([\alpha]M)[\alpha^* := E] &= [\alpha](M[\alpha^* := E])E \\ ([\beta]M)[\alpha^* := E] &= [\beta](M[\alpha^* := E]) \quad (\alpha \neq \beta) \\ (\mu\beta.M)[\alpha^* := E] &= \mu\beta.(M[\alpha^* := E]) \quad (\alpha \neq \beta) \\ (M[x.P, y.Q])[\alpha^* := E] &= (M[\alpha^* := E])[x.P[\alpha^* := E], y.Q[\alpha^* := E]] \end{aligned}$$

Definition 2.2 (Reduction rules). μ -reduction:

$$(\mu) \quad (\mu\alpha.M)E \rightarrow \mu\alpha.M[\alpha^* := E]$$

Permutative conversions:

$$(\pi\vee) \quad M[x.N, y.L]E \rightarrow M[x.NE, y.LE]$$

A context \mathcal{C} is defined in a usual way as a term with the hole:

$$\mathcal{C}, \mathcal{D} ::= \cdot|\mathcal{C}M|MC|[\alpha]\mathcal{C}|\mu\alpha.\mathcal{C}|\mathcal{C}[x.N, y.L]|M[x.\mathcal{C}, y.L]|M[x.N, y.\mathcal{C}]$$

$\mathcal{C}[M]$ is defined as a term obtained from \mathcal{C} by replacing the hole \cdot by M .

Congruency:

$$(congr) \quad \mathcal{C}[M] \rightarrow \mathcal{C}[N] \quad \text{if } M \rightarrow N.$$

The relation \rightarrow^* is defined as the reflexive transitive closure of the relation \rightarrow , and the relation \rightarrow^+ is defined as the transitive closure of the relation \rightarrow .

The redex of a reduction by the rule (μ) is $(\mu\alpha.M)E$. The redex of a reduction by the rule $(\pi\vee)$ is $M[x.N, y.L]E$. The redex of a reduction by the rule $(congr)$ is the redex of the reduction $M \rightarrow N$. In the redex $(\mu\alpha.M)E$, we call $\mu\alpha.M$ its function and E its argument respectively. In the redex $M[x.N, y.L]E$, we call $M[x.N, y.L]$ its function and E its argument respectively.

Remark. Church Rosser property holds.

A term M is defined to be strongly normalizing if there is no infinite reduction sequence $M \rightarrow M_1 \rightarrow M_2 \rightarrow \dots$ beginning with M .

It is well known that every term in this system is strongly normalizing [3].

For a reduction sequence $M \rightarrow M_1 \rightarrow \dots \rightarrow M_n$, its length is defined as n . $\|M\|$ is defined as the maximum length of reduction sequences beginning with M .

The next section will explicitly give us the maximum length $\|M\|$ by induction on the number of symbols in M .

3 Maximum Length of Reductions

This section will give the maximum length of reduction for a term by induction on the number of symbols in the term. For that, we will use a context-argument reduction tree.

We will use vector notation to denote a sequence. For example, \vec{M} denotes the sequence M_1, M_2, \dots, M_n and $M\vec{N}$ denotes $MN_1N_2\dots N_n$.

We define head constructors by $H ::= x\vec{M}|([\alpha]M)\vec{M}$. A head constructor does not have any function in its initial part. For example, $([\alpha]x)(\mu\beta.[\beta]y)z$ is a head constructor. On the other hand $([\alpha]x)[y_0.y_0, y_1.y_1]z$ is not a head constructor, since its initial part $([\alpha]x)[y_0.y_0, y_1.y_1]$ is a function.

A multicontext \mathcal{M} is defined by:

$$\mathcal{M} ::= \cdot|x|\mathcal{M}\mathcal{M}|[\alpha]\mathcal{M}|\mu\alpha.\mathcal{M}|\mathcal{M}[x.\mathcal{M}, x.\mathcal{M}].$$

A multicontext may have several holes and may not have any hole. $\mathcal{M}[N]$ is defined as a term obtained from \mathcal{M} by replacing all the hole \cdot by N . We will sometimes use \mathcal{C} and \mathcal{D} to denote a multicontext when we remark they are a multicontext.

Definition 3.1. The reduction is extended from a term to a context and a multicontext in a standard way by handling the hole \cdot as a fresh variable.

An eliminator context is defined by $\mathcal{E} ::= \mathcal{C}[[x.\mathcal{C}, y.M]]|[\alpha]M, y.\mathcal{C}$. An eliminator context is an eliminator with the hole. For example, $[x_0.y \cdot, x_1.x_1]$ is an eliminator context.

Context-argument reduction $\mathcal{M} \rightarrow_{ca} \mathcal{M}'$ is defined to hold if $\mathcal{M} \rightarrow \mathcal{M}'$ and the argument of its redex includes some hole. For example, $\mathcal{C} \rightarrow_{ca} \mathcal{M}$ holds if and only if $\mathcal{C} \rightarrow \mathcal{M}$ and its redex is $N\mathcal{E}$ where $\mathcal{C} = \mathcal{C}_1[N\mathcal{E}]$.

Non-context-argument reduction $\mathcal{M} \rightarrow_{nca} \mathcal{M}'$ is defined to hold if $\mathcal{M} \rightarrow \mathcal{M}'$ holds and $\mathcal{M} \rightarrow_{ca} \mathcal{M}'$ does not hold.

An effective redex is of the form either $M[x_0.N_0, x_1.N_1]L$ or $(\mu\alpha.M)N$ where α is in $\text{FN}(M)$.

Effective reduction $M \rightarrow_e M'$ is defined to hold if $M \rightarrow M'$ holds and its redex is effective.

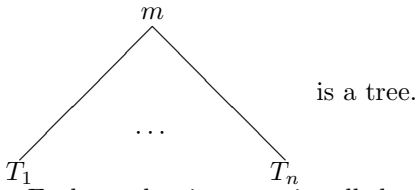
A null redex is of the form $(\mu\alpha.M)N$ where α is not in $\text{FN}(M)$.

Null reduction $M \rightarrow_n M'$ is defined to hold if $M \rightarrow M'$ holds and its redex is null.

A tree is defined as a finitely-branching tree with a natural number at each node.

Definition 3.2. A tree is defined as follows: (1) A natural number is a tree.

(2) If T_1, \dots, T_n are trees and m is a natural number, then

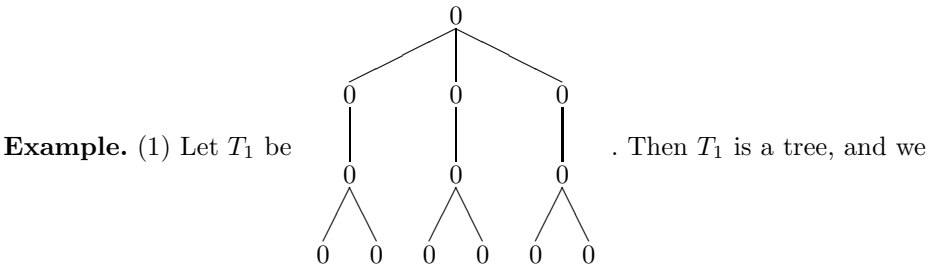


Each number in a tree is called a node. A lowermost node is called a leaf. We say an inner node to denote a node which is not a leaf. The uppermost node is called the root.

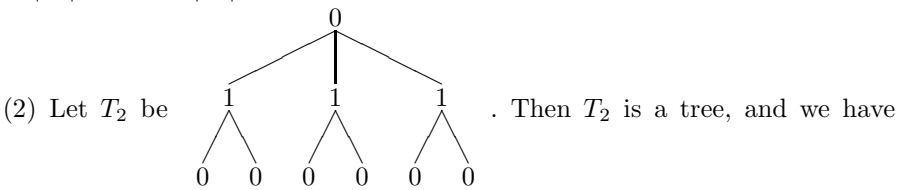
For a tree T , $|T|_l$ is the number of leaves in T and $|T|_n$ is defined as

(the number of inner nodes in T) + (the sum of all the nodes in T).

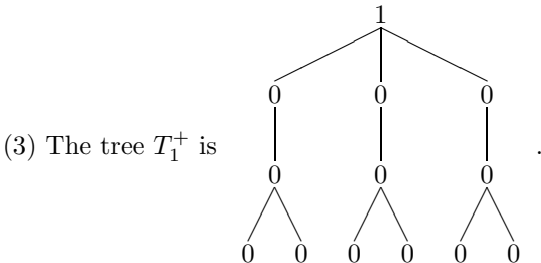
For a tree T , the tree T^+ is obtained from T by changing its root n to $n + 1$.



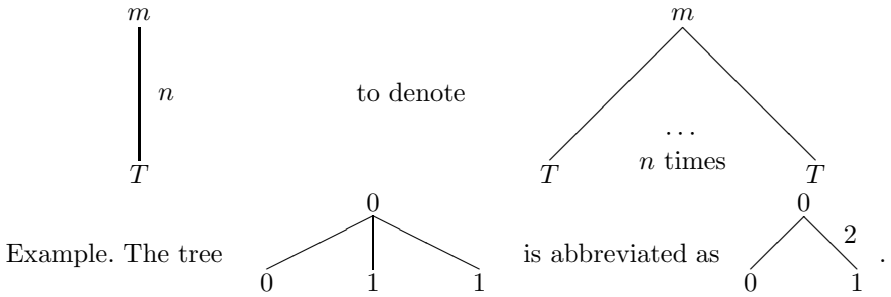
have $|T_1|_l = 6$ and $|T_1|_n = 7$.



$|T_2|_l = 6$ and $|T_2|_n = 7$.



Notation. We will use a branch with a natural number n as an abbreviation to denote n copies of the branch. That is, we write



A context-argument reduction tree $\text{Tr}(\mathcal{C})$ is a trace of context-argument reductions for \mathcal{C} . A parent node represents a subterm before reduction and its child nodes correspond to subterms produced by reduction. A null reduction is recorded by the number at the parent node. $\text{Tr}(\mathcal{C})$ is an invariant under effective non-context-argument reduction.

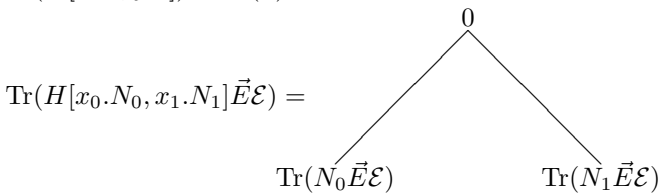
If all the reductions for \mathcal{C} are effective, each node in the tree $\text{Tr}(\mathcal{C})$ has the value 0. In this case, the number of inner nodes equals the maximum number of steps of context-argument reduction for \mathcal{C} . If we have some null context-argument reductions for \mathcal{C} , then some nodes in the tree $\text{Tr}(\mathcal{C})$ have a value n where $n > 0$. In this case, the sum $|\text{Tr}(\mathcal{C})|_n$ equals the maximum number of steps of context-argument reduction for \mathcal{C} . Note that a reduction step for \mathcal{C} is context-argument reduction if its redex is of the form $M\mathcal{E}$.

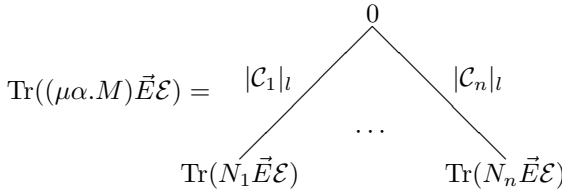
$|\text{Tr}(\mathcal{C})|_l$ equals the maximum number of copies of the hole by context-argument reduction.

Definition 3.3. $\#M$ is defined as the number of variable symbols and name symbols in the term M . For example, $\#(\mu\alpha.[\alpha]x)$ is 3. $\#\mathcal{C}$ is defined in the same way.

A context-argument reduction tree $\text{Tr}(\mathcal{C})$ of a context \mathcal{C} is defined by induction on $\#\mathcal{C}$ as follows where we write $|\mathcal{C}|_l$ and $|\mathcal{C}|_n$ for $|\text{Tr}(\mathcal{C})|_l$ and $|\text{Tr}(\mathcal{C})|_n$ respectively:

- $\text{Tr}(\cdot) = 0$
- $\text{Tr}(\mathcal{C}E) = \text{Tr}(\mathcal{C})$
- $\text{Tr}([\alpha]\mathcal{C}) = \text{Tr}(\mathcal{C})$
- $\text{Tr}(\mu\alpha.\mathcal{C}) = \text{Tr}(\mathcal{C})$
- $\text{Tr}(H\mathcal{C}) = \text{Tr}(\mathcal{C})$
- $\text{Tr}(H[x.\mathcal{C}, y.N]) = \text{Tr}(\mathcal{C})$
- $\text{Tr}(H[x.N, y.\mathcal{C}]) = \text{Tr}(\mathcal{C})$





where $\alpha \in \text{FN}(M)$ and $M = \mathcal{C}_i[[\alpha]N_i]$ ($1 \leq i \leq n$).

$\text{Tr}((\mu\alpha.M)\vec{E}\mathcal{E}) = \text{Tr}(x\vec{E}\mathcal{E})^+$ if $\alpha \notin \text{FN}(M)$.

Note that H is a head constructor, M, N are terms, and E is an eliminator.

Example

(1) Let \mathcal{C}_1 be $(\mu\alpha.x([\alpha]y)([\alpha]y)([\alpha]y))((\mu\beta.[\beta]y)((\mu\gamma.x([\gamma]y)([\gamma]y))\cdot))$. Then $\text{Tr}(\mathcal{C}_1) = T_1$ where the tree T_1 is given in the example after Definition 3.2. The maximum length of context-argument reduction for \mathcal{C}_1 is 7, which equals the number of inner nodes in this tree. The maximum number of copies of \cdot by reduction of \mathcal{C}_1 is 6, which equals the number of leaves in this tree.

(2) Let \mathcal{C}_2 be $(\mu\alpha.x([\alpha]y)([\alpha]y)([\alpha]y))((\mu\beta.y)((\mu\gamma.x([\gamma]y)([\gamma]y))\cdot))$. Then $\text{Tr}(\mathcal{C}_2) = T_2$ where the tree T_2 is given in the same example. The maximum length of context-argument reduction for \mathcal{C}_2 is 7, which equals the sum of the number of the inner nodes in this tree and the natural numbers at all the nodes. The maximum number of copies of \cdot by reduction of \mathcal{C}_2 is 6, which equals the number of leaves in this tree. Note that a null reduction by $\mu\beta.y$ is recorded by the natural number 1 at the corresponding nodes.

Definition 3.4. The measure $|M|$ is defined by

$$|M| = \Sigma_{M=C[N E]} |C|_l |N| \cdot |n|.$$

The next section will explain it and Section 5 will prove that $|M|$ is actually the maximum length of reduction for the term M .

4 Leftmost Short Reduction

This section gives intuitive explanation for $|M|$. For that, we will use leftmost short reduction and innermost null reduction, which produce a reduction sequence of the maximum length.

Definition 4.1 (Short redex). If the redex is either $(\mu\alpha.M)N$ with $\alpha \in \text{FN}(M)$, or $H[x_0.N_0, x_1.N_1]E$, or $(\mu\alpha.M)\vec{N}[x_0.N_0, x_1.N_1]E$ with $\alpha \notin \text{FN}(M)$, we call this redex a short redex. Note that M, N are terms, E is an eliminator, and H is a head constructor.

Short reduction $M \rightarrow_s M'$ is defined to hold if $M \rightarrow M'$ holds and its redex is short.

Note that a short redex is effective.

Example. In $u[v_0.v_0, v_1.v_1]x[y_0.y_0, y_1.y_1]z$, the short redex is $u[v_0.v_0, v_1.v_1]x$. In $(\mu\alpha.x([\alpha]y)([\alpha]y))x[y_0.y_0, y_1.y_1]z$, the short redex is $(\mu\alpha.x([\alpha]y)([\alpha]y))x$. In

$(\mu\alpha.x)x[y_0.y_0, y_1.y_1]z$, the short redex is $(\mu\alpha.x)x[y_0.y_0, y_1.y_1]z$. In the first and second examples, if we choose the whole term as a redex, we will miss the longest reduction sequence, since the redex can be copied and give twice steps. If we reduce a short redex, we can get the maximum number of copies of those redexes.

Definition 4.2 (Leftmost short reduction). Leftmost short reduction $M \rightarrow_{ls} M'$ is defined to hold if $M \rightarrow_s M'$ and its redex is leftmost among short redexes.

Example. In $u[v_0.v_0, v_1.v_1]x[y_0.y_0, y_1.y_1]((\mu\alpha.[\alpha]x)z)$, the short redexes are $u[v_0.v_0, v_1.v_1]x$ and $(\mu\alpha.[\alpha]x)z$. The leftmost short redex is $u[v_0.v_0, v_1.v_1]x$.

Note that if M has an effective redex, we have $M \rightarrow_{ls} M'$ for some M' , because some head part of the redex is a short redex and hence M has the leftmost short redex.

The next proposition shows that leftmost short reduction strategy gives us a reduction sequence of the maximum length. We can easily show this claim if we know it is strongly normalizing.

Proposition 4.3. *If M is strongly normalizing and $M \rightarrow_{ls} N$, then $\|M\| = \|N\| + 1$.*

Proof. We will show the claim by induction on $\|M\|$. Let $M \rightarrow M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_k$ be a reduction sequence of the maximum length.

Let M be $\mathcal{C}[LE]$ and LE be the redex for $M \rightarrow_{ls} N$. Suppose that in $M \rightarrow M_1$ the redex R is reduced to R' . Let M be $\mathcal{C}_1[R]$. Assume $M_1 \rightarrow_{ls} N_1$. We will show $N \rightarrow^+ N_1$ by considering cases according to R and LE .

Case 1. R is LEE_1 and E is $[y_0.P_0, y_1.P_1]$. If L is $H[x_0.L_0, x_1.L_1]$, then R' is $H[x_0.L_0, x_1.L_1][y_0.P_0E_1, y_1.P_1E_1]$, M_1 is $\mathcal{C}_1[R']$, and N is $\mathcal{C}_1[H[x_0.L_0E, x_1.L_1E]E_1]$. The leftmost short redex of M_1 is R' , so N_1 is $\mathcal{C}_1[H[x_0.L_0[y_0.P_0E_1, y_1.P_1E_1], x_1.L_1[y_0.P_0E_1, y_1.P_1E_1]]]$. So we have $N \rightarrow \mathcal{C}_1[H[x_0.L_0EE_1, x_1.L_1EE_1]] \rightarrow^* N_1$. If L is $(\mu\alpha.M)$ with $\alpha \in \text{FN}(M)$, the claim is proved similarly.

Case 2. R is in E . Then M_1 is $\mathcal{C}[LE']$ where $E \rightarrow E'$. If L is $H[x_0.L_0, x_1.L_1]$, then N is $\mathcal{C}[H[x_0.L_0E, x_1.L_1E]]$, and N_1 is $\mathcal{C}[H[x_0.L_0E', x_1.L_1E']]$, since the leftmost short redex of M_1 is LE' . So we have $N \rightarrow \mathcal{C}[H[x_0.L_0E', x_1.L_1E]] \rightarrow N_1$. If L is $(\mu\alpha.M)$ with $\alpha \in \text{FN}(M)$, the claim is proved similarly.

The other cases are proved in a similar way. Hence we have $N \rightarrow^+ N_1$.

Since $\|M\|$ is the maximum length and $M \rightarrow N$, we have $\|M\| \geq \|N\| + 1$. On the other hand, we have $\|M\| \leq \|N\| + 1$, since $\|N\| \geq \|N_1\| + 1 = \|M_1\| = \|M\| - 1$ by induction hypothesis $\|M_1\| = \|N_1\| + 1$. Therefore we have the claim. \square

Definition 4.4 (Innermost null reduction). Innermost null reduction $M \rightarrow_{in} M'$ is defined to hold if $M \rightarrow_n M'$ holds, M does not have any effective redex, and its redex is innermost.

If a term M does not have any effective redex, then the innermost null reduction strategy gives us a reduction sequence of the maximum length. The next proposition shows it.

Proposition 4.5. *If M is strongly normalizing, M does not have any effective redex, and $M \rightarrow_{in} N$, then $\|M\| = \|N\| + 1$.*

Proof. The claim is proved in a similar way to Proposition 4.3. \square

We will explain why $|M|$ gives k if we have a reduction sequence $M \rightarrow M_1 \rightarrow \dots \rightarrow M_k$ of the maximum length. First, we will group redexes used in the reduction sequence according to their origins. Counting the number of reduction steps is the same as counting the number of redexes chosen in each reduction step.

Every redex R in the reduction sequence has its origin R' in M . If a new redex R is obtained by copying a redex R' by reduction, then the origin of R is defined as R' . If a new redex R is created by reducing a redex R' , then the origin of R is defined as R' . For example, when $M_1 = (\mu\alpha.[\alpha]x)((\mu\beta.[\beta]y)z)$ reduces to $M_2 = \mu\alpha.[\alpha]x((\mu\beta.[\beta]y)z)$, then the origin of $(\mu\beta.[\beta]y)z$ in M_2 is $(\mu\beta.[\beta]y)z$ in M_1 . When $M_1 = (\mu\alpha.[\alpha](\mu\beta.[\beta]x))zw$ reduces to $M_2 = (\mu\alpha.[\alpha](\mu\beta.[\beta]x)z)w$, then the origin of $(\mu\beta.[\beta]x)z$ in M_2 is $(\mu\alpha.[\alpha](\mu\beta.[\beta]x)z)$ in M_1 .

All the redexes in the reduction sequence are grouped according to their origins. Let M be $\mathcal{C}[NE]$ and NE be a redex. Let X be the set of the redexes R such that R is chosen in the reduction sequence and the origin of R is NE . Let $|M|_{\mathcal{C}}$ be the number of elements in X . Then we have $\|M\| = \sum_{M=\mathcal{C}[NE]} |M|_{\mathcal{C}}$.

Secondly, we will evaluate each term $|M|_{\mathcal{C}}$. For simplicity, suppose M does not have any null redex. Let M be $\mathcal{C}[NE]$. Each term $|M|_{\mathcal{C}}$ is calculated by counting the steps of context-argument reduction for $\mathcal{C}[N\cdot]$ by leftmost short reduction. Hence we have $|M|_{\mathcal{C}} = |\mathcal{C}|_l |N \cdot|_n$. Remark that this description relies on the facts that (1) the number of copies of NE in M is determined only by \mathcal{C} and it does not depend on NE , and (2) the number of steps with redexes that come from NE is determined by only N and $|\mathcal{C}|_l$ and it does not depend on \mathcal{C} nor E . This property holds also when M has some null redexes.

Finally, combining these two equations, we have our description $|M| = \sum_{M=\mathcal{C}[NE]} |\mathcal{C}|_l |N \cdot|_n$. Note that this formula says that the maximum length for M is obtained by the sum of the maximum lengths of context-argument reduction sequences for each $\mathcal{C}[N\cdot]$. This is because counting the steps of context-argument reduction by leftmost short reduction is the same as counting the maximum steps of context-argument reduction.

5 Proof of Maximum Length

This section will prove that $|M|$ is the maximum length of reduction. This proof will also directly show that leftmost short reduction strategy and innermost null reduction strategy produce the longest reduction sequence.

We will fix a fresh variable \star . When a multicontext \mathcal{M} has n holes and $1 \leq i \leq n$, $\mathcal{M}^{(i)}$ denotes the context obtained from \mathcal{M} by substituting the fresh variable \star for all the holes except the i -th hole. For example, when $\mathcal{M} = x(y\cdot)(z\cdot)(w\cdot)$, we have $\mathcal{M}^{(1)} = x(y\cdot)(z\star)(w\star)$ and $\mathcal{M}^{(2)} = x(y\star)(z\cdot)(w\star)$.

Context-function reduction $\mathcal{C} \rightarrow_{cf} \mathcal{C}'$ is defined to hold if $\mathcal{C} \rightarrow \mathcal{C}'$ holds and its redex is of the form $\mathcal{C}_1 E$.

We will use notation like $M \rightarrow_{e, nca} M'$ to denote that $M \rightarrow_e M'$ and $M \rightarrow_{nca} M'$.

Lemma 5.1. (1) $Tr(\mathcal{C}[\alpha* := E]) = Tr(\mathcal{C})$.

(2) $|\mathcal{C}[\mathcal{E}]|_l = |\mathcal{C}|_l|\mathcal{E}|_l$.

(3) If $\mathcal{C} \rightarrow_e \mathcal{C}'$ or $\mathcal{C} \rightarrow_{cf} \mathcal{C}'$ holds and \mathcal{C}' has n holes, then $|\mathcal{C}'|_l = \sum_{i=1}^n |\mathcal{C}'^{(i)}|_l$.

(4) If $\mathcal{C} \rightarrow_{e, nca} \mathcal{C}'$, then $Tr(\mathcal{C}) = Tr(\mathcal{C}')$.

Proof. (1) By induction on $\sharp\mathcal{C}$.

(2) By induction on $\sharp\mathcal{C}$.

(3) By induction on $\sharp\mathcal{C}$. We will discuss several cases when \mathcal{C} is a redex. Cases 1 to 3 are for $\mathcal{C} \rightarrow_{cf} \mathcal{C}'$, and Cases 4 to 7 are for $\mathcal{C} \rightarrow_e \mathcal{C}'$. The other cases will be similarly shown by induction hypothesis.

Case 1. $(\mu\alpha.D)N \rightarrow \mu\alpha.D[\alpha* := N]$. The right-hand side $|\mathcal{C}'|_l$ equals $|\mathcal{D}[\alpha* := N]|_l$ by definition. By (1), it equals $|\mathcal{D}|_l$, which equals $|\mathcal{C}|_l$ by definition.

Case 2. $M[x_0.D, x_1.N_1]E \rightarrow M[x_0.DE, x_1.N_1E]$. The right-hand side $|\mathcal{C}'|_l$ equals $|M \cdot |l|\mathcal{D}|_l$ and $|\mathcal{C}|_l$ by definition and (2).

The case $M[x_0.N_0, x_1.D]E \rightarrow M[x_0.N_0E, x_1.DE]$ is proved similarly.

Case 3. $\mathcal{D}[x_0.N_0, x_1.N_1]E \rightarrow \mathcal{D}[x_0.N_0E, x_1.N_1E]$. By definition, the left-hand side and the right-hand side equal $|\mathcal{D}|_l$.

Case 4. $H[x_0.N_0, x_1.N_1]\vec{E}[y_0.L_0, y_1.L_1]\mathcal{E} \rightarrow H[x_0.N_0, x_1.N_1]\vec{E}[y_0.L_0\mathcal{E}, y_1.L_1\mathcal{E}]$. The right-hand side is $|H[x_0.N_0, x_1.N_1]\vec{E}[y_0.L_0\mathcal{E}, y_1.L_1\mathcal{E}[\star]]|_l + |H[x_0.N_0, x_1.N_1]\vec{E}[y_0.L_0\mathcal{E}[\star], y_1.L_1\mathcal{E}]|_l$, which equals $|N_0\vec{E}[y_0.L_0\mathcal{E}, y_1.L_1\mathcal{E}[\star]]|_l + |N_1\vec{E}[y_0.L_0\mathcal{E}, y_1.L_1\mathcal{E}[\star]]|_l + |N_0\vec{E}[y_0.L_0\mathcal{E}[\star], y_1.L_1\mathcal{E}]|_l + |N_1\vec{E}[y_0.L_0\mathcal{E}[\star], y_1.L_1\mathcal{E}]|_l$ by definition. By induction hypothesis, it equals $|N_0\vec{E}[y_0.L_0, y_1.L_1]\mathcal{E}|_l + |N_1\vec{E}[y_0.L_0, y_1.L_1]\mathcal{E}|_l$, which equals the left-hand side by definition.

Case 5. $H[x_0.N_0, x_1.N_1]\mathcal{E} \rightarrow H[x_0.N_0\mathcal{E}, x_1.N_1\mathcal{E}]$. This case is proved similarly to Case 7.

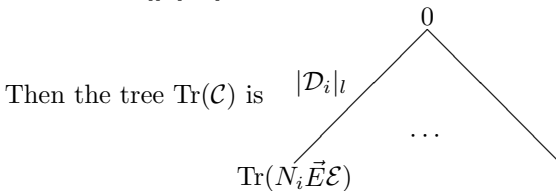
Case 6. $(\mu\alpha.M)\vec{E}[x_0.N_0, x_1.N_1]\mathcal{E} \rightarrow (\mu\alpha.M)\vec{E}[x_0.N_0\mathcal{E}, x_1.N_1\mathcal{E}]$. This case is proved similarly to Case 4.

Case 7. $(\mu\alpha.M)\mathcal{E} \rightarrow \mu\alpha.M[\alpha* := \mathcal{E}]$. Let $M = \mathcal{D}_i[[\alpha]N_i]$. We write \mathcal{D}'_i and N'_i for $\mathcal{D}_i[\alpha* := \mathcal{E}[\star]]$ and $N_i[\alpha* := \mathcal{E}[\star]]$ respectively. $\mathcal{C}'^{(i)}$ is $\mu\alpha.\mathcal{D}'_j[[\alpha]N'_j\mathcal{E}]$ for some j . The right-hand side $\sum_i |\mathcal{C}'^{(i)}|_l$ equals $\sum_i |\mathcal{D}'_i[[\alpha]N'_i\mathcal{E}]|_l$. By (2), each term $|\mathcal{D}'_i[[\alpha]N'_i\mathcal{E}]|_l$ equals $|\mathcal{D}'_i|_l|N'_i\mathcal{E}|_l$. By (1), it equals $|\mathcal{D}_i|_l|N_i\mathcal{E}|_l$. Hence we have $\sum_i |\mathcal{C}'^{(i)}|_l = \sum_i |\mathcal{D}_i|_l|N_i\mathcal{E}|_l$, which equals $|\mathcal{C}|_l$ by the definition of $Tr((\mu\alpha.M)\mathcal{E})$.

(4) By induction on $\sharp\mathcal{C}$. We will consider cases according to $\mathcal{C} \rightarrow \mathcal{C}'$.

Case 1. $(\mu\alpha.M)\vec{E}\mathcal{E} \rightarrow (\mu\alpha.M')\vec{E}\mathcal{E}$.

Let $M = \mathcal{D}_i[[\alpha]N_i]$ for $1 \leq i \leq n$.



We will show that each branch for $Tr(N_i\vec{E}\mathcal{E})$ will be preserved by the reduction.

Case 1.1. The redex is outside $[\alpha]N_i$. Then we have $M' = \mathcal{D}'_i[[\alpha]N_i]$ and $\mathcal{D}_i \rightarrow_e \mathcal{D}'_i$. By (3), we get $|\mathcal{D}'_i|_l = |\mathcal{D}_i|_l$. Hence the branch is preserved.

Case 1.2. The function part of the redex includes $[\alpha]N_i$. Then we have $M' = \mathcal{D}'_i[[\alpha]N'_i]$, $\mathcal{D}_i \rightarrow_{cf} \mathcal{D}'_i$, and $N'_i = N_i[\beta* := L]$ for some β and L . By (3), we have $|\mathcal{D}'_i|_l = |\mathcal{D}_i|_l$. By (1), we have $\text{Tr}(N'_i \vec{E}\mathcal{E}) = \text{Tr}(N_i \vec{E}\mathcal{E})$. Hence the branch is preserved.

Case 1.3. The argument part of the redex includes $[\alpha]N_i$. Then we have $M' = \mathcal{D}'_i[[\alpha]N_i]$ and $\mathcal{D}_i \rightarrow_e \mathcal{D}'_i$. Note that \mathcal{D}'_i may have more than one holes. By (3), we have $|\mathcal{D}'_i|_l = \sum_j |\mathcal{D}'_i^{(j)}|_l$. Hence the branch is preserved.

Case 1.4. The redex is included in $[\alpha]N_i$. Then we have $M' = \mathcal{D}_i[[\alpha]N'_i]$ and $N_i \rightarrow_e N'_i$. Hence we have $N_i \vec{E}\mathcal{E} \rightarrow_{e, nca} N'_i \vec{E}\mathcal{E}$. By induction hypothesis, we have $\text{Tr}(N_i \vec{E}\mathcal{E}) = \text{Tr}(N'_i \vec{E}\mathcal{E})$. Hence the branch is preserved.

Case 2. $(\mu\alpha.M)\vec{E}[y_0.\mathcal{D}, y_1.L_1]E \rightarrow (\mu\alpha.M)\vec{E}[y_0.DE, y_1.L_1E]$. Let $M = \mathcal{C}_i[[\alpha]N_i]$. A child node of $\text{Tr}(\mathcal{C})$ is $\text{Tr}(N_i \vec{E}[y_0.\mathcal{D}, y_1.L_1]E)$. By induction hypothesis, it equals $\text{Tr}(N_i \vec{E}[y_0.DE, y_1.L_1E])$, which gives a child node of $\text{Tr}(\mathcal{C}')$.

Case 3. $(\mu\alpha.\mathcal{D})E \rightarrow \mu\alpha.\mathcal{D}[\alpha* := E]$. We have $\text{Tr}(\mathcal{C}) = \text{Tr}(\mathcal{D})$ by definition. By (1), it equals $\text{Tr}(\mathcal{D}[\alpha* := E])$, which equals $\text{Tr}(\mathcal{C}')$.

The other cases are proved in a similar way to Cases 2 and 3. \square

Lemma 5.2. (1) $|ME \cdot |_n = |M \cdot |_l |xE \cdot |_n + |M \cdot |_n$ holds.

(2) $|M[x_0.N_0, x_1.N_1] \cdot |_n \geq 1 + |M \cdot |_l (|N_0 \cdot |_n + |N_1 \cdot |_n)$ holds.

Proof. (1) Induction on $\sharp M$.

Case 1. $M = H$ and $E = N$ for some head constructor H and some term N . The left-hand side equals $|HN \cdot |_n = 0$ by definition, which equals the right-hand side since $|H \cdot |_n = 0$ and $|xN \cdot |_n = 0$.

Case 2. $M = H$ for some head constructor H and $E = [x_0.N_0, x_1.N_1]$. The left-hand side equals $1 + |N_0 \cdot |_n + |N_1 \cdot |_n$ by definition, which equals the right-hand side since $|H \cdot |_l = 1$, $|H \cdot |_n = 0$, and $|x[x_0.N_0, x_1.N_1] \cdot |_n = 1 + |N_0 \cdot |_n + |N_1 \cdot |_n$.

Case 3. $M = H[y_0.N_0, y_1.N_1]\vec{E}$. By definition the left-hand side equals $1 + |N_0 \vec{E}E \cdot |_n + |N_1 \vec{E}E \cdot |_n$. By induction hypothesis, it equals $1 + |N_0 \vec{E} \cdot |_l |xE \cdot |_n + |N_0 \vec{E} \cdot |_n + |N_1 \vec{E} \cdot |_l |xE \cdot |_n + |N_1 \vec{E} \cdot |_n$. By definition it equals the right-hand side since $|H[x_0.N_0, x_1.N_1]\vec{E} \cdot |_l = |N_0 \vec{E} \cdot |_l + |N_1 \vec{E} \cdot |_l$ and $|H[x_0.N_0, x_1.N_1]\vec{E} \cdot |_n = 1 + |N_0 \vec{E} \cdot |_n + |N_1 \vec{E} \cdot |_n$.

Case 4. $M = (\mu\alpha.L)\vec{E}$ and $\alpha \in \text{FN}(L)$. Let L be $\mathcal{C}_i[[\alpha]N_i]$ ($1 \leq i \leq n$). By definition the left-hand side is $1 + \sum_{i=1}^n |\mathcal{C}_i|_l |N_i \vec{E}E \cdot |_n$. By induction hypothesis it equals $1 + \sum_{i=1}^n |\mathcal{C}_i|_l (|N_i \vec{E} \cdot |_l |xE \cdot |_n + |N_i \vec{E} \cdot |_n)$. It equals the left-hand side, since $|(\mu\alpha.L)\vec{E} \cdot |_l = \sum_i |\mathcal{C}_i|_l |N_i \vec{E} \cdot |_l$ and $|(\mu\alpha.L)\vec{E} \cdot |_n = 1 + \sum_i |\mathcal{C}_i|_l |N_i \vec{E} \cdot |_n$.

Case 5. $M = (\mu\alpha.L)\vec{E}$ and $\alpha \notin \text{FN}(L)$. The claim is proved similarly to Case 4.

(2) By letting E be $[x_0.N_0, x_1.N_1]$ in (1), we have the right-hand side $|M \cdot |_l (1 + |N_0 \cdot |_n + |N_1 \cdot |_n) + |M \cdot |_n$, which is equal to or greater than $1 + |M \cdot |_l (|N_0 \cdot |_n + |N_1 \cdot |_n)$, since $|M \cdot |_l \geq 1$ and $|M \cdot |_n \geq 0$. \square

Definition 5.3. Suppose $\mathcal{C}[NE]$ reduces to M with its redex R , R is not NE , and $N'E'$ is a redex in M . We say $N'E'$ is inherited from NE when (1) R is in

\mathcal{C} , the reduction does not change NE , and $N'E'$ is NE , (2) the reduction copies NE and $N'E'$ is a copy of it, (3) R is inside either N or E , and NE reduces to $N'E'$, or (4) R is $(\mu\alpha.P)Q$, NE is inside P , and $N'E'$ is $(NE)[\alpha* := Q]$.

The next lemma guarantees that each term $|\mathcal{C}|_l|N \cdot |_n$ is unchanged under effective reduction if its redex is not NE .

Lemma 5.4. *If $\mathcal{C}[NE] \rightarrow_e \mathcal{M}[N'E']$ holds, its redex is not NE , and $N'E'$ is inherited from NE , then $|\mathcal{C}|_l = \Sigma_i |\mathcal{M}^{(i)}|_l$ and $|N \cdot |_n = |N' \cdot |_n$ hold.*

Proof. Let the redex of the reduction be R . Let R be PF for some term P and some eliminator F . We will consider cases according to R and NE .

Case 1. R is outside NE or F includes NE . Then we have $N' = N$ and $\mathcal{C} \rightarrow_e \mathcal{M}$. By Lemma 5.1 (3), we have $|\mathcal{C}|_l = \Sigma_i |\mathcal{M}^{(i)}|_l$. Hence we have the claim.

Case 2. P includes NE . Then we have $\mathcal{C} \rightarrow_{cf} \mathcal{M}$ and $N' = N[\alpha* := L]$ for some α and L . By Lemma 5.1 (3), we have $|\mathcal{C}|_l = |\mathcal{M}|_l$. By Lemma 5.1 (1), we get $|N'|_n = |N|_n$.

Case 3. N includes R . Then we have $\mathcal{M} = \mathcal{C}$. Since $N \cdot \rightarrow_{e, nca} N' \cdot$, By Lemma 5.1 (4), we have $|N \cdot |_n = |N' \cdot |_n$.

Case 4. E includes R . Then we have $\mathcal{M} = \mathcal{C}$ and $N' = N$. □

The next proposition says that $|M|$ decreases by effective reduction.

Proposition 5.5. (1) *If $M \rightarrow_s M'$ holds with the redex R and $M = \mathcal{C}[R]$, we have $|M| = |M'| + |\mathcal{C}|_l$.*

(2) *If $M \rightarrow_e M'$ holds with the redex R and $M = \mathcal{C}[R]$, we have $|M| \geq |M'| + |\mathcal{C}|_l$.*

Proof. (1) The definition gives us

$$\begin{aligned} |M| &= \Sigma_{M=\mathcal{D}[NE]} |\mathcal{D}|_l |N \cdot |_n, \\ |M'| &= \Sigma_{M'=\mathcal{D}'[N'E']} |\mathcal{D}'|_l |N' \cdot |_n. \end{aligned}$$

Let R be the redex PF for some term P and some eliminator F .

All the redexes in M' are (a) those inherited from all the redexes in M except R and (b) new redexes $P_j F_j$ created by the reduction.

For (a), Lemma 5.4 gives $|\mathcal{D}|_l |N \cdot |_n = \Sigma_i |\mathcal{M}^{(i)}|_l |N' \cdot |_n$ for each redex $N'E'$ in M' inherited from the redex NE in M where $\mathcal{D} \rightarrow \mathcal{M}$ or $\mathcal{D} = \mathcal{M}$ and $M' = \mathcal{M}[N'E']$.

For (b), we will show $|\mathcal{C}|_l |P \cdot |_n = |\mathcal{C}|_l + \Sigma_i |\mathcal{C}_i|_l |P_i \cdot |_n$ where $M' = \mathcal{C}_i [P_i F_i]$. Let $R \rightarrow R'$. We will consider cases according to P .

Case 1. $P = H[x_0.N_0, x_1.N_1]$ for some head constructor H . We have $R' = H[x_0.N_0F, x_1.N_1F]$. Then the right-hand side is $|\mathcal{C}|_l + |\mathcal{C}[H[x_0. \cdot, x_1.N_1F]]|_l |N_0 \cdot |_n + |\mathcal{C}[H[x_0.N_0F, x_1. \cdot]]|_l |N_1 \cdot |_n$, which equals $|\mathcal{C}|_l + |\mathcal{C}|_l |N_0 \cdot |_n + |\mathcal{C}|_l |N_1 \cdot |_n$ by definition. By the definition of the tree, we have $|H[x_0.N_0, x_1.N_1] \cdot |_n = 1 + |N_0 \cdot |_n + |N_1 \cdot |_n$. Hence the left-hand side $|\mathcal{C}|_l |P \cdot |_n$ equals the right-hand side.

Case 2. $P = \mu\alpha.S$ and $\alpha \in \text{FN}(S)$. We have $R' = \mu\alpha.S[\alpha* := F]$. Let $S = \mathcal{C}_i[[\alpha]Q_i]$ for $1 \leq i \leq n$. Let \mathcal{C}'_i and Q'_i be $\mathcal{C}_i[\alpha* := F]$ and $Q_i[\alpha* := F]$ respectively. Then the right-hand side is $|\mathcal{C}|_l + \sum_i |\mathcal{C}[\mu\alpha.\mathcal{C}'_i[[\alpha]\cdot]]|_l |Q'_i \cdot|_n$. By Lemma 5.1 (2), it equals $|\mathcal{C}|_l + \sum_i |\mathcal{C}|_l |\mathcal{C}'_i|_l |Q'_i \cdot|_n$. By Lemma 5.1 (1), it equals $|\mathcal{C}|_l + \sum_i |\mathcal{C}|_l |\mathcal{C}_i|_l |Q_i \cdot|_n$. By the definition of the tree, we have $|(\mu\alpha.S) \cdot|_n = 1 + \sum_i |\mathcal{C}_i|_l |Q_i \cdot|_n$. Hence the left-hand side $|\mathcal{C}|_l |P \cdot|_n$ equals the right-hand side.

Case 3. $P = (\mu\alpha.M)\vec{N}[x_0.N_0, x_1.N_1]$ and $\alpha \notin \text{FN}(M)$. We have $R' = (\mu\alpha.M)\vec{N}[x_0.N_0F, x_1.N_1F]$. This case is proved similarly to Case 1.

(2) This claim is proved similarly to (1). For (a), the same equation is proved in the same way. For (b), instead we will prove $|\mathcal{C}|_l |P \cdot|_n \geq |\mathcal{C}|_l + \sum_i |\mathcal{C}_i|_l |P_i \cdot|_n$ where $M' = \mathcal{C}_i[P_iF_i]$. Instead of Cases 1 and 3 in (1), we have the next case.

Case 1. $P = M[x_0.N_0, x_1.N_1]$. We have $R' = M[x_0.N_0F, x_1.N_1F]$. This case is proved by using Lemma 5.2 (2). \square

The eliminator E is called an effective argument in M if M has a subterm of the form either $(\mu\alpha.M)\vec{E}E$ with $\alpha \in \text{FN}(M)$ or $N[x_0.N_0, x_1.N_1]\vec{E}E$. Then E will be used as an argument of some effective reduction.

Lemma 5.6. *If the hole is not in any effective argument in \mathcal{C} , then $\text{Tr}(\mathcal{C}) = n$ holds for some n .*

Proof. By induction on $\sharp\mathcal{C}$. We will discuss only a non-trivial case.

Case 1. $\mathcal{C} = (\mu\alpha.M)\vec{N}\mathcal{C}_1$ and $\alpha \notin \text{FN}(M)$. By the definition, we have $\text{Tr}(\mathcal{C}) = \text{Tr}(\mathcal{C}_1)^+$. By induction hypothesis, we have $\text{Tr}(\mathcal{C}_1) = n$ for some n . Hence $\text{Tr}(\mathcal{C}) = n + 1$ holds. \square

The next theorem states main property for leftmost short reduction.

Theorem 5.7. *If $M \rightarrow_{ls} M'$ holds, then $|M| = |M'| + 1$ holds.*

Proof. Let $M = \mathcal{C}[R] \rightarrow_{ls} M' = \mathcal{C}[R']$ and R be the redex. Since R is the leftmost short redex in M , the hole is not in any effective argument in \mathcal{C} . By Lemma 5.6, we have $|\mathcal{C}|_l = 1$. By Theorem 5.5 (1), we get the claim. \square

The next theorem says that $|M|$ decreases by null reduction.

Theorem 5.8. (1) *If $M \rightarrow_n M'$ holds, then $|M| \geq |M'| + 1$ holds.*

(2) *If $M \rightarrow_{in} M'$ holds, then $|M| = |M'| + 1$ holds.*

Proof. (1) Let $M = \mathcal{C}[(\mu\alpha.N)L]$ and $M' = \mathcal{C}[\mu\alpha.N]$. By definition we have $|M| - |M'| = |\mathcal{C}|_l |(\mu\alpha.N) \cdot|_n + \sum_{L=\mathcal{D}[PQ]} |\mathcal{C}[(\mu\alpha.N)\mathcal{D}]|_l |P \cdot|_n \geq |\mathcal{C}|_l |(\mu\alpha.N) \cdot|_n$. By definition, we have $\text{Tr}((\mu\alpha.N)\cdot) = 1$. Hence we have $|\mathcal{C}|_l |(\mu\alpha.N) \cdot|_n \geq 1$ and we get the claim.

(2) In a similar way to (1), we have $|M| - |M'| = |\mathcal{C}|_l |(\mu\alpha.N) \cdot|_n$, since the redex $(\mu\alpha.N)L$ is innermost and L does not include any redex. By Lemma 5.6, we have $|\mathcal{C}|_l = 1$. So the right-hand side equals 1 and we have the claim. \square

Proposition 5.9. *If $M \rightarrow N$, then $|M| > |N|$.*

Proof. If $M \rightarrow N$ is effective, the claim follows from Proposition 5.5 (2). If $M \rightarrow N$ is null, we have the claim from Theorem 5.8 (1). \square

Theorem 5.10. $|M|$ is the maximum length of reduction steps for a term M .

Proof. First we will show that $|M|$ is an upper bound of any length of reduction steps. If $M \rightarrow M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_n$ holds, then by Proposition 5.9, we have $|M| > |M_1| > |M_2| > \dots > |M_n| \geq 0$. Therefore we have $|M| \geq n$.

Secondly, we will show $|M|$ is the maximum by induction on $|M|$. To do that, we will construct a reduction sequence beginning with M with $|M|$ steps. If $|M| = 0$, we have the reduction sequence M of length 0, so we have the claim. Suppose $|M| > 0$. By definition of $|M|$, M has some redex.

We have M' such that $M \rightarrow M'$ and $|M| = |M'| + 1$. This is shown as follows. If M has an effective redex, we have $M \rightarrow_{ls} M'$ for some M' . By Theorem 5.7, $|M| = |M'| + 1$ holds. If M does not have any effective redex, there is a null redex. We have $M \rightarrow_{in} M'$ for some M' . By Theorem 5.8 (2), $|M| = |M'| + 1$ holds.

By induction hypothesis for M' , we have a reduction sequence $M' \rightarrow \dots \rightarrow N$ with $|M'|$ steps. Hence we have a reduction sequence $M \rightarrow M' \rightarrow \dots \rightarrow N$ with $|M|$ steps. □

This proof gives us the following reduction strategy for choosing a redex to be reduced in a term M .

1. Leftmost short reduction. If M has an effective redex, then some head part of the redex is a short redex, so we choose the leftmost short redex.
2. Innermost null reduction. If M does not have any effective redex, we choose an innermost null redex.

This proof also showed that this strategy actually gives the longest reduction sequence.

6 Concluding Remarks

The results of this paper are based on the property that the maximum steps for each redex can be calculated independently of the other redexes, and the maximum steps for a term can be obtained as the sum of the maximum steps for all the redexes.

Future work will be applying this technique to $\mu\mu'$ -reduction in symmetric $\lambda\mu$ -calculus.

Another future work will be giving a small upper bound for μ -reduction steps by using the exact count achieved in this paper. We had $|M| = \sum_{M=C[NE]} |C|_l |N \cdot |n$. By easy estimation, the number of applications NE is less than $\sharp M$, and we have $|C|_l \leq 2^{\sharp M}$ and $|N \cdot |n \leq 2^{(\sharp M)^2/2}$. So we have $|M| \leq \sharp M \cdot 2^{\sharp M + (\sharp M)^2/2}$. Further investigation will improve this upper bound.

Acknowledgments

We would like to thank Prof. Philippe de Groote for valuable discussions and comments. We would also like to thank Dr. Koji Nakazawa for plenty of discussions and interest on this subject.

References

1. David, R., Nour, K.: A short proof of the strong normalization of classical natural deduction with disjunction. *Journal of Symbolic Logic* 68(4), 1277–1288 (2003)
2. David, R., Nour, K.: Arithmetical Proofs of Strong Normalization Results for the Symmetric $\lambda\mu$ -Calculus. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 162–178. Springer, Heidelberg (2005)
3. de Groote, P.: Strong normalization of classical natural deduction with disjunction. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, pp. 182–196. Springer, Heidelberg (2001)
4. de Groote, P.: On the strong normalisation of intuitionistic natural deduction with permutation-conversions. *Information and Computation* 178, 441–464 (2002)
5. Joachimski, F., Matthes, R.: Short proofs of normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic* 42, 59–87 (2003)
6. Matthes, R.: Non-strictly positive fixed-points for classical natural deduction. *Annals of Pure and Applied Logic* 133(1–3), 205–230 (2005)
7. Nakazawa, K., Tatsuta, M.: Strong normalization proof with CPS-translation for second order classical natural deduction. *Journal of Symbolic Logic* 68(4), 851–859 (2003)
8. Nour, K., Saber, K.: A semantical proof of the strong normalization theorem for full propositional classical natural deduction. *Archive for Mathematical Logic* 45(3), 357–364 (2006)
9. Parigot, M.: Strong normalization for second order classical natural deduction. *Journal of Symbolic Logic* 62(4), 1461–1479 (1997)
10. Prawitz, D.: *Natural Deduction*. Almqvist and Wiksell (1965)
11. Tatsuta, M., Mints, G.: A simple proof of second-order strong normalization with permutative conversions. *Annals of Pure and Applied Logic* 136(1–2), 134–155 (2005)

On Linear Combinations of λ -Terms

Lionel Vaux

Institut de Mathématiques de Luminy, CNRS UMR 6206, France
vaux@iml.univ-mrs.fr

Abstract. We define an extension of λ -calculus with linear combinations, endowing the set of terms with a structure of \mathbf{R} -module, where \mathbf{R} is a fixed set of scalars. Terms are moreover subject to identities similar to usual pointwise definition of linear combinations of functions with values in a vector space. We then extend β -reduction on those algebraic λ -terms as follows: $at + u$ reduces to $at' + u$ as soon as term t reduces to t' and a is a non-zero scalar. We prove that reduction is confluent.

Under the assumption that the set \mathbf{R} of scalars is positive (*i.e.* a sum of scalars is zero iff all of them are zero), we show that this algebraic λ -calculus is a conservative extension of ordinary λ -calculus. On the other hand, we show that if \mathbf{R} admits negative elements, then every term reduces to every other term.

Preliminary Definitions and Notations. Recall that a rig (or “semi-ring with zero and unit”) is the same as a ring, without the condition that every element admits an opposite for addition. Let \mathbf{R} be a rig. We write \mathbf{R}^\bullet for $\mathbf{R} \setminus \{0\}$. We denote by letters a, b, c the elements of \mathbf{R} , and say that \mathbf{R} is positive if, for all $a, b \in \mathbf{R}$, $a + b = 0$ implies $a = 0$ and $b = 0$. An example of positive rig is \mathbf{N} , the set of natural numbers, with usual operations. Also, we write application of λ -terms à la Krivine: $(s)t$ denotes the application of term s to term t .

1 Introduction

Sums of terms arise naturally in the study of differentiation in λ -calculus [ER03] or $\lambda\mu$ -calculus [Vau07]. In this setting, non-deterministic choice provides a possible computational interpretation of sum. In differential λ -calculus, however, a more general pattern is introduced: the set of terms is endowed with a structure of \mathbf{R} -module, where \mathbf{R} is a commutative rig, and one can form linear combinations of terms. Moreover, in the same way as functions with values in a vector space also form a vector space with operations defined pointwise, we have the following two equalities on terms:

$$\lambda x \left(\sum_{i=1}^n a_i s_i \right) = \sum_{i=1}^n a_i \lambda x s_i \quad \text{and} \quad \left(\sum_{i=1}^n a_i s_i \right) u = \sum_{i=1}^n a_i (s_i) u \quad (1)$$

for all linear combination $\sum_{i=1}^n a_i s_i$ of terms. This mimics the quantitative semantics of λ -calculus in finiteness spaces [Ehr05]: types are interpreted by particular vector spaces or, more generally, modules, and terms are mapped to analytic functions defined by power series on these spaces.

Apart from the notion of differentiation, one important feature of the above-mentioned works is the way β -reduction is extended to such linear combinations of terms. Among terms, some are considered simple: they contain no sum in linear position, so that (II) does not apply; hence they are intrinsically not sums. These form a basis of the \mathbb{R} -module of terms. Reduction \rightarrow is then the least contextual relation such that: if s is a simple term, then

$$(\lambda x s) t \rightarrow s [t/x] \tag{2}$$

and, if $a \in \mathbb{R}^\bullet$ is a non-zero scalar,

$$s \rightarrow s' \text{ implies } as + t \rightarrow as' + t . \tag{3}$$

The condition $a \neq 0$ in that last case ensures that \rightarrow actually reduces something, so that reduction is not trivially reflexive.

The previous definition is both natural in presence of coefficients, and technically efficient. For instance, it is particularly well suited for proving confluence via usual Tait-Martin L of technique: introduce a parallel version \rightarrow of \rightarrow such that $\rightarrow \subseteq \rightarrow \subseteq \rightarrow^*$, and prove that \rightarrow has the diamond property. Here \rightarrow is reflexive and has the following behaviour on linear combinations of terms:

$$\sum_{i=1}^n a_i s_i \rightarrow \sum_{i=1}^n a_i s'_i \text{ as soon as, for all } i, s_i \rightarrow s'_i \text{ and } s_i \text{ is simple.} \tag{4}$$

Assuming $s \rightarrow s' \rightarrow s''$ are simple terms, we have $s + s' \rightarrow 2s'$ and $s + s' \rightarrow s + s''$: then (4) allows to close that pair of reductions by $2s' \rightarrow s' + s''$ and $s + s'' \rightarrow s' + s''$. This would not hold if we had forced the s_i 's in (4) to be distinct simple terms — that condition would amount to reduce each element of the base of simple terms, in parallel, which may seem a more natural choice at first.

In [Vau07], however, the author proved that this notion of reduction collapses as soon as the rig of scalars admits negative elements: if $-1 \in \mathbb{R}$ (so that $1 + (-1) = 0$), then for all terms s and t , $s \rightarrow^* t$. This should not be a surprise, since in that case the system involves both negative numbers and potential infinity through arbitrary fixed points: in that setting, one can't hope to implement (II) and still obtain both confluence and coherence of β -reduction.

Also, our notion of reduction does not always fit well with normalization: assume $s \rightarrow s'$ and \mathbb{R} contains dyadic rationals; then

$$s = \frac{1}{2}s + \frac{1}{2}s \rightarrow \frac{1}{2}s + \frac{1}{2}s' \rightarrow \frac{1}{4}s + \frac{3}{4}s' \rightarrow \dots$$

Contributions. In this paper, we give a framework for the study of terms with linear combinations, which aims to be more precise and formal than that developed in [ER03] or [Vau07]. Also, we do not consider differentiation nor classical control operators, and only focus on the algebraic structure of terms and the interaction between coefficients and reduction. We call the obtained system algebraic λ -calculus.

In section 2, we formalize the definition of the \mathbb{R} -module of terms; in particular, we implement the identities of (1), orienting them from left to right; then we identify terms up to equality of canonical forms. This definition is elementary enough that it should be easily implemented in a logical system such as Coq (assuming an implementation of \mathbb{R}). In section 3 we define reduction, using rule (3) in the case of a sum, and discuss conservativity w.r.t. ordinary β -reduction. In section 4, we briefly review sufficient conditions for normalization: the reader may refer to [Vau06] for a full development on that matter. Last, we discuss possible other approaches and further work in section 5.

Most of the results of this paper were already present in [Vau07], and some can be traced back to [ER03]. In those two previous works, however, the focus was on differentiation and the presence of linear combinations of terms and their effects on reduction were considered of marginal interest. This may in particular explain why some of the problems we insist on in this paper eluded [ER03].

2 Linear Combinations of Terms

In this section, we introduce the set of terms of algebraic λ -calculus in several steps. First we give a grammar of terms, on which we define α -equivalence and substitution as in Krivine's [Kri90]. Then we define canonical forms of terms; this endows the set of terms with a structure of module, by identifying terms up to equality of canonical forms.

2.1 Raw Terms

Let be given a denumerable set V of variables. We use letters among x, y, z to denote variables.

Definition 1. *The set $L_{\mathbb{R}}$ of raw terms (denoted by greek letters σ, τ, \dots) of algebraic λ -calculus over \mathbb{R} is given by the following grammar:*

$$\sigma, \tau, \dots ::= x \mid \lambda x \sigma \mid (\sigma) \tau \mid \mathbf{0} \mid a\sigma \mid \sigma + \tau .$$

Definition 2. *We define free variables of terms as follows:*

- variable x is free in term y if $x = y$;
- variable x is free in $\lambda y \sigma$ if $x \neq y$ and x is free in σ ;
- variable x is free in $(\sigma) \tau$ if x is free in σ or in τ ;
- no variable is free in $\mathbf{0}$;
- variable x is free in $a\sigma$ if x is free in σ ;
- variable x is free in term $\sigma + \tau$ if x is free in σ or in τ .

From this definition of free variables, we derive α -equivalence and substitution as in [Kri90]. We write $\sigma \sim \tau$ when σ is α -convertible to τ , and we write $\sigma[\tau/x]$ for the (capture-avoiding) substitution of τ for x in σ . More generally, if x_1, \dots, x_n are distinct variables and τ_1, \dots, τ_n are terms, we write $\sigma[\tau_1, \dots, \tau_n/x_1, \dots, x_n]$ for the simultaneous substitution of each τ_i for each x_i in σ . Recall the following definitions and properties from [Kri90].

Proposition 1. For all terms $\sigma, \tau_1, \dots, \tau_n, v_1, \dots, v_p$ and all distinct variables $x_1, \dots, x_n, y_1, \dots, y_p$,

$$\begin{aligned} & \sigma [\tau_1, \dots, \tau_n / x_1, \dots, x_n] [v_1, \dots, v_p / y_1, \dots, y_p] \\ & \sim \sigma [v_1, \dots, v_p, \tau'_1, \dots, \tau'_n / y_1, \dots, y_p, x_1, \dots, x_n] \end{aligned}$$

where $\tau'_i = \tau_i [v_1, \dots, v_p / y_1, \dots, y_p]$.

Definition 3. A binary relation r on raw terms is said to be contextual if it satisfies the following conditions:

- $x \ r \ x$;
- $\lambda x \ \sigma \ r \ \lambda x \ \sigma'$ as soon as $\sigma \ r \ \sigma'$;
- $(\sigma) \ \tau \ r \ (\sigma') \ \tau'$ as soon as $\sigma \ r \ \sigma'$ and $\tau \ r \ \tau'$;
- $\mathbf{0} \ r \ \mathbf{0}$;
- $a \ \sigma \ r \ a \ \sigma'$ as soon as $\sigma \ r \ \sigma'$;
- $\sigma + \tau \ r \ \sigma' + \tau'$ as soon as $\sigma \ r \ \sigma'$ and $\tau \ r \ \tau'$.

Proposition 2. If r is a contextual relation, then $\sigma [\tau / x] \ r \ \sigma [\tau' / x]$ as soon as $\tau \ r \ \tau'$.

Proposition 3. Relation \sim is a contextual equivalence relation.

2.2 Permutative Equality

If $\sigma_1, \dots, \sigma_n \in L_R$, then we write $\sigma_1 + \dots + \sigma_n$ for $\sigma_1 + (\dots + \sigma_n)$. If, moreover, $a_1, \dots, a_n \in R$ then we write $\sum_{i=1}^n a_i \sigma_i$ for the term $a_1 \sigma_1 + \dots + a_n \sigma_n + \mathbf{0}$. Linear combinations $\sum_{i=1}^n a_i \sigma_i$ should be thought of as multisets of couples, *i.e.* we identify $\sum_{i=1}^n a_i \sigma_i$ with all $\sum_{i=1}^n a_{f(i)} \sigma_{f(i)}$ where f is any permutation of $\{1, \dots, n\}$. This is more formally stated in the following definition.

Definition 4. Permutative equality $\equiv \subseteq L_R \times L_R$ is the least contextual equivalence relation on raw terms such that:

- $\sigma \equiv \tau$ as soon as $\sigma \sim \tau$;
- $\sigma + \tau \equiv \tau + \sigma$ for all $\sigma, \tau \in L_R$;
- $(\sigma + \tau) + v \equiv \sigma + \tau + v$ for all $\sigma, \tau, v \in L_R$.

Notice that $\sum_{i=1}^n a_i \sigma_i \equiv \sum_{j=1}^p b_j \tau_j$ iff $n = p$ and, for all j , $b_j = a_{f(j)}$ and $\tau_j \equiv \sigma_{f(j)}$, with f some fixed permutation of $\{1, \dots, n\}$. Also, since free variables of a sum do not depend on the order of the summands, \equiv preserves free variables.

Permutative equality is the basic equality intended on terms. It states that we consider terms up to α -equality and that we form linear combinations up to associativity and commutativity of sum. We write Λ_R for the set of terms with equality \equiv . This means that as long we consider σ and τ as terms in Λ_R , we say they are equal if $\sigma \equiv \tau$. A function defined on Λ_R is a function with domain L_R which is invariant by \equiv .

Proposition 4. Substitution is well defined on Λ_R : if $\sigma \equiv \sigma'$ and $\tau_i \equiv \tau'_i$, for all $i \in \{1, \dots, n\}$, then $\sigma [\tau_1, \dots, \tau_n / x_1, \dots, x_n] \equiv \sigma' [\tau'_1, \dots, \tau'_n / x_1, \dots, x_n]$ for all distinct variables x_1, \dots, x_n .

In the following, if σ and $\tau \in \Lambda_R$, we write $\delta_{\sigma, \tau} \equiv \begin{cases} 1 & \text{if } \sigma \equiv \tau \\ 0 & \text{otherwise.} \end{cases}$

2.3 The R-Module of Terms

In this subsection, we introduce the algebraic content of the calculus: we endow the set of terms with a structure of \mathbf{R} -module, enjoying usual identities between linear combinations together with **(II)**. For that purpose, we define canonical forms of terms, so that equality of terms (in the abovementioned algebraic sense) amounts to permutative equality on canonical forms.

Definition 5. *Atomic terms and canonical terms are defined as follows:*

- any variable x is an atomic term;
- let $x \in \mathbf{V}$ and s an atomic term, then $\lambda x s$ is an atomic term;
- let s an atomic term and T a canonical term, then $(s)T$ is an atomic term;
- let $a_1, \dots, a_n \in \mathbf{R}^\bullet$ and s_1, \dots, s_n n pairwise distinct (\neq) atomic terms, then $\sum_{i=1}^n a_i s_i$ is a canonical term.

We consider atomic and canonical terms up to permutative equality. We write $A_{\mathbf{R}}$ for the set of atomic terms and $C_{\mathbf{R}}$ for the set of canonical terms, both endowed with \equiv as identity relation. One defines an injection from atomic terms into canonical terms, mapping s to the “singleton” $1s + \mathbf{0}$. In the following, we write σ for term $1\sigma + \mathbf{0}$.

Definition 6. *Let $\sigma = \sum_{i=1}^n a_i s_i$ be a linear combination of atomic terms. For all atomic term s , we call coefficient of s in σ the scalar $\sum_{i=1}^n \delta_{s, s_i}^{\equiv} a_i$. Then we define*

$$\text{cansum}(\sigma) = \sum_{j=1}^p b_j t_j$$

where $\{t_1, \dots, t_p\}$ is the set (modulo \equiv) of those s_i ’s with a non-zero coefficient in σ and, for all $j \in \{1, \dots, p\}$, b_j is the coefficient of t_j in σ .

Hence, if σ is a linear combination of atomic terms, then $\text{cansum}(\sigma)$ is a canonical term.

Definition 7. *Canonization of terms $\text{can} : A_{\mathbf{R}} \longrightarrow C_{\mathbf{R}}$ is given by*

- $\text{can}(x) = x$;
- if $\text{can}(\sigma) = \sum_{i=1}^n a_i s_i$ then $\text{can}(\lambda x \sigma) = \sum_{i=1}^n a_i \lambda x s_i$;
- if $\text{can}(\sigma) = \sum_{i=1}^n a_i s_i$ and $\text{can}(\tau) = T$ then $\text{can}((\sigma)\tau) = \sum_{i=1}^n a_i (s_i)T$;
- $\text{can}(\mathbf{0}) = \mathbf{0}$;
- if $\text{can}(\sigma) = \sum_{i=1}^n a_i s_i$ then $\text{can}(a\sigma) = \text{cansum}(\sum_{i=1}^n (aa_i) s_i)$;
- if $\text{can}(\sigma) = \sum_{i=1}^n a_i s_i$ and $\text{can}(\tau) = \sum_{i=n+1}^{n+p} a_i s_i$ then

$$\text{can}(\sigma + \tau) = \text{cansum} \left(\sum_{i=1}^{n+p} a_i s_i \right) .$$

It is easily checked that this definition is invariant by \equiv .

Proposition 5. *Canonization enjoys the following properties.*

- (i) Variables free in $\text{can}(\sigma)$ are also free in σ . The converse does not hold in general.
- (ii) If s is an atomic term, then $\text{can}(s) \equiv s$.
- (iii) If S is a canonical term, then $\text{can}(S) \equiv S$; hence $\text{can}(\text{can}(\sigma)) \equiv \text{can}(\sigma)$ for all term σ .
- (iv) For all terms σ and τ and all variable x ,

$$\text{can}(\sigma [\tau/x]) \equiv \text{can}(\text{can}(\sigma) [\text{can}(\tau)/x]) .$$

Definition 8. Algebraic equality is permutative equality of canonical forms:

$$\sigma \stackrel{\nabla}{=} \tau \quad \text{if} \quad \text{can}(\sigma) \equiv \text{can}(\tau) .$$

Although it does not preserve free variables, algebraic equality is a contextual equivalence relation. Restricted to canonical terms, it is the same as \equiv .

Definition 9. A simple term is a term σ such that $\text{can}(\sigma) = s$ with s atomic, or equivalently such that there exists an atomic term s with $s \stackrel{\nabla}{=} \sigma$. We write $\Delta_{\mathbf{R}}$ for the set of simple terms, with equality $\stackrel{\nabla}{=}$ and $\mathbf{R}\langle\Delta_{\mathbf{R}}\rangle$ for the set of all terms, with equality $\stackrel{\nabla}{=}$, which is the free \mathbf{R} -module generated by $(\Delta_{\mathbf{R}}, \stackrel{\nabla}{=})$.

Lemma 1. If terms σ, σ', τ and τ' are such that $\sigma \stackrel{\nabla}{=} \sigma'$ and $\tau \stackrel{\nabla}{=} \tau'$, then, for all variable x , $\sigma [\tau/x] \stackrel{\nabla}{=} \sigma' [\tau'/x]$.

Proof. This is a straightforward application of Proposition [5](#).

Hence, algebraic equality is compatible with substitution, *i.e.* substitution is well defined on $\mathbf{R}\langle\Delta_{\mathbf{R}}\rangle$. We now extend the notion of coefficient as follows:

Definition 10. Let σ be a simple term and s an atomic term such that $\sigma \stackrel{\nabla}{=} s$. We define the coefficient of σ in τ , denoted by $\tau_{(\sigma)}$, as the coefficient of s in $\text{can}(\tau)$.

We call support of σ the set of those simple terms with a non-zero coefficient in σ :

$$\text{Supp}(\sigma) = \{ \tau \in \Delta_{\mathbf{R}}; \sigma_{(\tau)} \neq 0 \} .$$

Definition 11. If \mathcal{X} is a set (modulo $\stackrel{\nabla}{=}$) of simple terms, we write $\mathbf{R}\langle\mathcal{X}\rangle$ for the set of linear combinations of elements of \mathcal{X} , *i.e.*

$$\mathbf{R}\langle\mathcal{X}\rangle = \left\{ \sigma \stackrel{\nabla}{=} \sum_{i=1}^n a_i \sigma_i; \forall i \in \{1, \dots, n\}, \sigma_i \in \mathcal{X} \right\}$$

or, equivalently,

$$\mathbf{R}\langle\mathcal{X}\rangle = \{ \sigma \in \mathbf{R}\langle\Delta_{\mathbf{R}}\rangle; \text{Supp}(\sigma) \subseteq \mathcal{X} \} .$$

Remark 1. One may introduce a convergent rewrite system (modulo associativity and commutativity of sum [PSS1]) R on terms in $\Delta_{\mathbb{R}}$, such that $\sigma \stackrel{\forall}{=} \tau$ iff $\text{NF}(\sigma) \equiv \text{NF}(\tau)$ where NF stands for “normal form in R ”. For instance, one may adapt the work by Arrighi and Dowek in [AD05], provided the rig \mathbb{R} is appropriately defined by a *scalar rewrite system*.

Apart from the added complexity, we do not use this approach simply because such a rewrite system R would not be part of the reduction rules we introduce thereafter: we define reduction of terms as (roughly) β -reduction up to $\stackrel{\forall}{=}$, and not as the union of β -reduction and canonization. We then prove confluence without any assumption on \mathbb{R} .

Of course, our approach may suffer from practical drawbacks: unless we have suitable constructive information on the structure of \mathbb{R} , we cannot compute all the $\widetilde{\rightarrow}$ -reducts of a term in general (we actually prove that there may be an infinity of them in some cases). Also, checking that $\sigma \rightarrow \tau$, or even $\sigma \stackrel{\forall}{=} \tau$, requires at least that equality in \mathbb{R} is decidable.

3 Reductions

In this section, we define reduction using (2) and (3) as key reduction rules: this captures the definition of reduction in [ER03], minus differentiation, in the setting of algebraic λ -calculus.

3.1 Reduction and Linear Combinations of Terms

We call algebraic relation from simple terms to terms any subset of $\Delta_{\mathbb{R}} \times \mathbb{R}\langle \Delta_{\mathbb{R}} \rangle$ (with, of course, the condition that it is invariant under $\stackrel{\forall}{=}$). Clearly, if r is an algebraic relation from simple terms to terms, $\sigma \ r \ \sigma'$ holds iff there are $t \in \Delta_{\mathbb{R}}$ and $T' \in \mathbb{C}_{\mathbb{R}}$ such that $\sigma \stackrel{\forall}{=} t$, $\sigma' \stackrel{\forall}{=} T'$ and $t \ r \ T'$.

Similarly, we call algebraic relation from terms to terms any subset of $\mathbb{R}\langle \Delta_{\mathbb{R}} \rangle \times \mathbb{R}\langle \Delta_{\mathbb{R}} \rangle$. Again such a relation is uniquely defined by its restriction to $\mathbb{C}_{\mathbb{R}} \times \mathbb{C}_{\mathbb{R}}$. Given an algebraic relation r from simple terms to terms we define two new algebraic relations \bar{r} and \tilde{r} from terms to terms by:

- $\sigma \ \bar{r} \ \sigma'$ if $\sigma \stackrel{\forall}{=} \sum_{i=1}^n a_i s_i$ and $\sigma' \stackrel{\forall}{=} \sum_{i=1}^n a_i S'_i$, where for all $i \in \{1, \dots, n\}$, s_i is atomic, S'_i is canonical and $s_i \ r \ S'_i$;
- $\sigma \ \tilde{r} \ \sigma'$ if $\sigma \stackrel{\forall}{=} at + U$ and $\sigma' \stackrel{\forall}{=} aT' + U$, where $a \neq 0$, t is atomic, T' and U are canonical and $t \ r \ T'$.

We cannot define reduction by induction on terms: if there are $a, b \in \mathbb{R}^\bullet$ such that $a + b = 0$ then $\mathbf{0} \stackrel{\forall}{=} a\sigma + b\sigma$ for all $\sigma \in \mathbb{R}\langle \Delta_{\mathbb{R}} \rangle$; hence, by rule (3), $\mathbf{0}$ may reduce. We rather define simple term reduction \rightarrow by induction on the depth of the fired redex, so that reduction of terms is given by $\widetilde{\rightarrow}$.

Definition 12. *We define an increasing sequence of algebraic relations from simple terms to terms by the following statements. \rightarrow_0 is the empty relation. Assume \rightarrow_k is defined. Then we set $\sigma \rightarrow_{k+1} \sigma'$ as soon as one of the following holds:*

- $\sigma \stackrel{\nabla}{=} \lambda x s$ and $\sigma' \stackrel{\nabla}{=} \lambda x S'$ with $s \rightarrow_k S'$;
- $\sigma \stackrel{\nabla}{=} (s) T$ and $\sigma' \stackrel{\nabla}{=} (S') T$ with $s \rightarrow_k S'$, or $\sigma' \stackrel{\nabla}{=} (s) T'$ with $T \xrightarrow{\sim}_k T'$;
- $\sigma \stackrel{\nabla}{=} (\lambda x s) T$ and $\sigma' \stackrel{\nabla}{=} s [T/x]$.

Let $\rightarrow = \bigcup_{k \in \mathbf{N}} \rightarrow_k$. We call one-step reduction or simply reduction, the algebraic relation $\xrightarrow{\sim}$.

Proposition 6. $\xrightarrow{\sim} = \bigcup_{k \in \mathbf{N}} \xrightarrow{\sim}_k$.

Lemma 2. If $s \in A_{\mathbf{R}}$ and $S', T, T' \in C_{\mathbf{R}}$, are such that $s \rightarrow S'$ and $T \xrightarrow{\sim} T'$ then:

$$\begin{aligned} \lambda x s &\rightarrow \lambda x S' \\ (s) T &\rightarrow (S') T \\ (s) T &\rightarrow (s) T' . \end{aligned}$$

Proof. The first two relations are straightforward from the definition of \rightarrow . The same holds for the third one, through proposition 6.

Let $\xrightarrow{\sim}^*$ be the reflexive and transitive closure of $\xrightarrow{\sim}$.

Lemma 3. The relation $\xrightarrow{\sim}^*$ is contextual.

Proof. This results from Lemma 2, using reflexivity, transitivity and the definition of can.

3.2 Confluence

We prove confluence of $\xrightarrow{\sim}$ by usual Tait-Martin-Löf technique: introduce a parallel extension of reduction (in which redexes can be fired simultaneously) and prove this enjoys the diamond property (*i.e.* strong confluence).

Definition 13. We define an increasing sequence of algebraic relations from simple terms to terms by the following statements. \rightarrow_0 is algebraic equality. Assume \rightarrow_k is defined. Then we set $\sigma \rightarrow_{k+1} \sigma'$ as soon as one of the following holds:

- $\sigma \stackrel{\nabla}{=} \lambda x s$ and $\sigma' \stackrel{\nabla}{=} \lambda x S'$ with $s \rightarrow_k S'$;
- $\sigma \stackrel{\nabla}{=} (s) T$ and $\sigma' \stackrel{\nabla}{=} (S') T'$ with $s \rightarrow_k S'$ and $T \xrightarrow{\rightarrow}_k T'$;
- $\sigma \stackrel{\nabla}{=} (\lambda x s) T$ and $\sigma' \stackrel{\nabla}{=} S' [T'/x]$ with $s \rightarrow_k S'$ and $T \xrightarrow{\rightarrow}_k T'$.

Let $\rightarrow = \bigcup_{k \in \mathbf{N}} \rightarrow_k$. We call parallel reduction the algebraic relation $\xrightarrow{\rightarrow}$.

Proposition 7. $\xrightarrow{\rightarrow} = \bigcup_{k \in \mathbf{N}} \xrightarrow{\rightarrow}_k$.

Lemma 4. Relation $\xrightarrow{\rightarrow}$ is contextual.

Proof. Like in Lemma 2, this is just rephrasing the definitions of π and $\bar{\pi}$, with the notable exception of the application case which involves Proposition 7.

Lemma 5. $(\lambda x \sigma) \tau \xrightarrow{\rightarrow} \sigma' [\tau'/x]$ as soon as $\sigma \xrightarrow{\rightarrow} \sigma'$ and $\tau \xrightarrow{\rightarrow} \tau'$.

Proof. This is a straightforward consequence of Lemma 4 and the definitions of $\xrightarrow{\rightarrow}$ and $\text{can}(\lambda x \sigma)$.

Lemma 6. $\xrightarrow{\sim} \subset \xrightarrow{\rightarrow} \subset \xrightarrow{\sim}^*$.

Proof. $\xrightarrow{\sim} \subset \xrightarrow{\rightarrow}$ should be clear. $\xrightarrow{\rightarrow} \subset \xrightarrow{\sim}^*$ follows from contextuality of $\xrightarrow{\sim}^*$.

Reductions and Substitution. The main property of parallel reduction is the following one, which fails for one-step reduction.

Lemma 7. *Let x be a variable and $\sigma, \tau, \sigma', \tau'$ be terms. If $\sigma \twoheadrightarrow \sigma'$ and $\tau \twoheadrightarrow \tau'$ then*

$$\sigma [\tau/x] \twoheadrightarrow \sigma' [\tau'/x] .$$

Proof. We prove by induction on k that if $\sigma \twoheadrightarrow_k \sigma'$ and $\tau \twoheadrightarrow \tau'$ then $\sigma [\tau/x] \twoheadrightarrow \sigma' [\tau'/x]$. If $k = 0$ then $\sigma' \stackrel{\forall}{=} \sigma \stackrel{\forall}{=} \text{can}(\sigma)$; then by Lemmas 1 and 4, and Proposition 2, we have

$$\sigma [\tau/x] \stackrel{\forall}{=} \text{can}(\sigma) [\tau/x] \twoheadrightarrow \text{can}(\sigma) [\tau'/x] \stackrel{\forall}{=} \sigma' [\tau'/x] .$$

Suppose the result holds for some k , then we extend it to $k + 1$ by inspecting the possible cases for reduction $\sigma \twoheadrightarrow_{k+1} \sigma'$. We first address the case in which σ is simple and $\sigma \rightarrow_{k+1} \sigma'$. Then one of the following statements applies:

- $\sigma \stackrel{\forall}{=} \lambda y t$ with $y \neq x$ and y not free in τ , and $\sigma' \stackrel{\forall}{=} \lambda y T'$ with $t \rightarrow_k T'$; hence, by induction hypothesis, $t [\tau/x] \twoheadrightarrow T' [\tau'/x]$ and we get

$$\sigma [\tau/x] \stackrel{\forall}{=} \lambda y t [\tau/x] \twoheadrightarrow \lambda y T' [\tau'/x] \stackrel{\forall}{=} \sigma' [\tau'/x]$$

by Lemma 4

- $\sigma \stackrel{\forall}{=} (t) V$ and $\sigma' \stackrel{\forall}{=} (T') V'$ with $t \rightarrow_k T'$ and $V \twoheadrightarrow_k V'$: by induction hypothesis, $t [\tau/x] \twoheadrightarrow T' [\tau'/x]$ and $V [\tau/x] \twoheadrightarrow V' [\tau'/x]$ and we get

$$\sigma [\tau/x] \stackrel{\forall}{=} (t [\tau/x]) V [\tau/x] \twoheadrightarrow (T' [\tau'/x]) V' [\tau'/x] \stackrel{\forall}{=} \sigma' [\tau'/x]$$

by Lemma 4

- $\sigma \stackrel{\forall}{=} (\lambda y t) V$ and $\sigma' \stackrel{\forall}{=} T' [V'/y]$ with $t \rightarrow_k T'$, $V \twoheadrightarrow_k V'$, $x \neq y$ and y not free in τ : by induction hypothesis, $t [\tau/x] \twoheadrightarrow T' [\tau'/x]$, $V [\tau/x] \twoheadrightarrow V' [\tau'/x]$ and we get

$$\sigma [\tau/x] \stackrel{\forall}{=} (\lambda y t [\tau/x]) V [\tau/x] \twoheadrightarrow (T' [\tau'/x]) [V' [\tau'/x]/y] \stackrel{\forall}{=} \sigma' [\tau'/x] .$$

by Lemma 5

Now assume $\sigma \twoheadrightarrow_{k+1} \sigma'$. By definition, this amounts to the following: $\sigma \stackrel{\forall}{=} \sum_{i=1}^n a_i s_i$ and $\sigma' \stackrel{\forall}{=} \sum_{i=1}^n a_i S'_i$, with $s_i \rightarrow_{k+1} S'_i$ for all i . We have just shown that we then have $s_i [\tau/x] \twoheadrightarrow S'_i [\tau'/x]$. We conclude by lemma 4

From Lemmas 6 and 7, we can derive a very similar result for $\widetilde{\twoheadrightarrow}^*$:

Corollary 1. *Let x be a variable and $\sigma, \tau, \sigma', \tau'$ be terms. If $\sigma \widetilde{\twoheadrightarrow}^* \sigma'$ and $\tau \widetilde{\twoheadrightarrow}^* \tau'$ then*

$$\sigma [\tau/x] \widetilde{\twoheadrightarrow}^* \sigma' [\tau'/x] .$$

Church-Rosser. We finish the proof of confluence by showing that the \rightarrow -reducts of a fixed term σ all \rightarrow -reduce to one of them (obtained by firing all redexes of σ , simultaneously).

Definition 14. We define inductively on canonical term S its full parallel reduct $S\downarrow$ by:

$$\begin{aligned} x\downarrow &\stackrel{\forall}{=} x \\ (\lambda x s)\downarrow &\stackrel{\forall}{=} \lambda x s\downarrow \\ ((\lambda x s)T)\downarrow &\stackrel{\forall}{=} (s\downarrow)[T\downarrow/x] \\ ((s)T)\downarrow &\stackrel{\forall}{=} (s\downarrow)T\downarrow \text{ if } s \text{ is a variable or an application} \\ \left(\sum_{i=1}^n a_i s_i\right)\downarrow &\stackrel{\forall}{=} \sum_{i=1}^n a_i s_i\downarrow . \end{aligned}$$

For all term σ , we set $\sigma\downarrow \stackrel{\forall}{=} \text{can}(\sigma)\downarrow$.

Lemma 8. If σ and σ' are such that $\sigma \overrightarrow{=} \sigma'$, then $\sigma' \overrightarrow{=} \sigma\downarrow$.

Proof. One simply proves by induction on k that if $\sigma \overrightarrow{=}_k \sigma'$ then $\sigma' \overrightarrow{=} \sigma\downarrow$, using Lemma 7.

Theorem 1. Relation $\overrightarrow{=}$ is strongly confluent. Hence, relation $\widetilde{=}$ enjoys the Church-Rosser property.

Proof. Strong confluence of $\overrightarrow{=}$ is a straightforward corollary of lemma 8. It implies confluence of $\widetilde{=}$ by Lemma 6.

Trivial. There is a case in which confluence is much easier to establish: if 1 admits an opposite $-1 \in R$. In this case, assume $\sigma \widetilde{=}^* \sigma'$. Since $\widetilde{=}^*$ is algebraic and contextual, $\sigma' \stackrel{\forall}{=} \sigma' + (-1)\sigma + \sigma \widetilde{=}^* \sigma' + (-1)\sigma' + \sigma \stackrel{\forall}{=} \sigma$. Hence $\widetilde{=}^*$ is symmetric, which obviously implies Church-Rosser. But this has little meaning: in the next section, we show that reduction becomes trivial as soon as $-1 \in R$.

3.3 Conservativity

Notice that every ordinary λ -term is also a simple term of algebraic λ -calculus. Let Λ denote the set of all λ -terms and $\rightarrow_\beta \subset \Lambda \times \Lambda$ the usual β -reduction of λ -calculus. It is clear that $\rightarrow_\beta \subset \hookrightarrow$.

Denote by \leftrightarrow the reflexive, symmetric and transitive closure of $\widetilde{=}$ and \leftrightarrow_β the usual β -equivalence of λ -calculus.

Lemma 9. Algebraic λ -calculus preserves the equalities of λ -calculus, i.e. for all λ -terms s and t , $s \leftrightarrow_\beta t$ implies $s \leftrightarrow t$.

Proof. This is a straightforward consequence of the confluence of \rightarrow_β and the fact that $\rightarrow_\beta \subset \widetilde{=}$.

One may wonder if the reverse also holds, *i.e.* if equivalence classes of λ -terms in algebraic λ -calculus are the same as in ordinary λ -calculus. If R is \mathbf{N} , then $\widetilde{\rightarrow}$ -reductions from λ -terms are exactly \rightarrow_β -reductions ($\stackrel{\vee}{\rightarrow}$ only amounts to α -conversion on λ -terms), and the result holds by the same argument as in Lemma 9. In the general case, however, a λ -term does not necessarily reduce to another λ -term, hence the proof is not as easy.

The Positive Case. In the following, we prove that $\leftrightarrow \cap (\Lambda \times \Lambda) = \leftrightarrow_\beta$ as soon as R is positive.

Definition 15. We define $\Lambda : C_R \rightarrow \mathcal{P}(\Lambda)$ by the following statements:

$$\begin{aligned} \Lambda(x) &= \{x\} \\ \Lambda(\lambda x s) &= \{\lambda x u; u \in \Lambda(s)\} \\ \Lambda((s)T) &= \{(u)v; u \in \Lambda(s) \text{ and } v \in \Lambda(T)\} \\ \Lambda\left(\sum_{i=1}^n a_i s_i\right) &= \bigcup_{i=1}^n \Lambda(s_i) . \end{aligned}$$

For all term σ , we set $\Lambda(\sigma) = \Lambda(\text{can}(\sigma))$.

Proposition 8. If $s \in \Lambda$, then $\Lambda(s) = \{s\}$.

Lemma 10. If R is positive and terms $\sigma \in \Lambda_R$ and $\sigma' \in \Lambda_R$ are such that $\sigma \widetilde{\rightarrow} \sigma'$, then for all $s' \in \Lambda(\sigma')$, either $s' \in \Lambda(\sigma)$ or there exists $s \in \Lambda(\sigma)$ such that $s \rightarrow_\beta s'$.

Proof. The proof is by induction on the height of the reduction $\sigma \widetilde{\rightarrow} \sigma'$. All induction steps are straightforward, except for the extension from \rightarrow_k to $\widetilde{\rightarrow}_k$: assume $\sigma \stackrel{\vee}{\rightarrow} at + U$ and $\sigma' \stackrel{\vee}{\rightarrow} aT' + U$ with $a \neq 0$ and $t \rightarrow_k T'$. By definition, $\Lambda(\sigma') = \Lambda(aT' + U) \subseteq \Lambda(T') \cup \Lambda(U)$. Moreover, since R is positive, the coefficient of t in $at + U$ is non-zero: hence $\Lambda(\sigma) = \Lambda(at + U) = \Lambda(t) \cup \Lambda(U)$. Now assume $v' \in \Lambda(\sigma')$: either $v' \in \Lambda(U) \subset \Lambda(\sigma)$; or $v' \in \Lambda(T')$, and then, by induction hypothesis, either $v' \in \Lambda(t) \subset \Lambda(\sigma)$ or there exists $v \in \Lambda(t) \subset \Lambda(\sigma)$ such that $v \rightarrow v'$.

Corollary 2. If R is positive and $s \in \Lambda$ and $\sigma \in R\langle \Delta_R \rangle$ are such that $s \widetilde{\rightarrow}^* \sigma$, then for all $t \in \Lambda(\sigma)$, $s \rightarrow_\beta^* t$.

Lemma 11. If σ and $\sigma' \in R\langle \Delta_R \rangle$ are such that $\sigma \overrightarrow{\rightarrow} \sigma'$ then $\sigma \downarrow \overrightarrow{\rightarrow} \sigma' \downarrow$.

Proof. The proof is easy and very close to that of Lemma 8.

We define iterated full reduction by $\sigma \downarrow^0 \stackrel{\vee}{=} \sigma$ and $\sigma \downarrow^{n+1} \stackrel{\vee}{=} (\sigma \downarrow^n) \downarrow$.

Lemma 12. If $\sigma \overrightarrow{\rightarrow}^n \tau$ then $\tau \widetilde{\rightarrow}^* \sigma \downarrow^n$.

Proof. The proof is by induction on n . If $n = 0$, $\sigma \stackrel{\vee}{\rightarrow} \tau \stackrel{\vee}{\rightarrow} \sigma \downarrow^0$ and this is reflexivity of $\widetilde{\rightarrow}^*$. Assume the result holds at rank n . If $\sigma \overrightarrow{\rightarrow}^n \tau \overrightarrow{\rightarrow} \tau'$, then, by induction hypothesis, $\tau \widetilde{\rightarrow}^* \sigma \downarrow^n$. Since $\widetilde{\rightarrow}^*$ is also the transitive closure of $\overrightarrow{\rightarrow}$, Lemma 11 entails $\tau \downarrow \widetilde{\rightarrow}^* \sigma \downarrow^{n+1}$. By Lemma 8, we have $\tau' \overrightarrow{\rightarrow} \tau \downarrow$, hence $\tau' \widetilde{\rightarrow}^* \sigma \downarrow^{n+1}$.

Theorem 2. *If R is positive and $s, t \in \Lambda$ are such that $s \leftrightarrow t$ then $s \leftrightarrow_{\beta} t$.*

Proof. Assume $s, t \in \Lambda$ and $s \leftrightarrow t$. By the Church-Rosser property of $\widetilde{\rightarrow}$ (Theorem 11), there exists $\sigma \in R\langle \Delta_R \rangle$ such that $s \widetilde{\rightarrow}^* \sigma$ and $t \widetilde{\rightarrow}^* \sigma$. By Lemma 12, there exists some $n \in \mathbf{N}$ such that $\sigma \widetilde{\rightarrow}^* v = s \downarrow^n$. Notice that if $w \in \Lambda$, then $w \downarrow \in \Lambda$, hence $v \in \Lambda$. We have $s \widetilde{\rightarrow}^* v$ and $t \widetilde{\rightarrow}^* v$, hence by positivity of R and Corollary 2, for all $v' \in \Lambda(v)$ there are $s' \in \Lambda(s)$ and $t' \in \Lambda(t)$ such that $s' \rightarrow_{\beta}^* v'$ and $t' \rightarrow_{\beta}^* v'$. By proposition 8, $\Lambda(s) = \{s\}$, $\Lambda(t) = \{t\}$ and $\Lambda(v) = \{v\}$, hence the conclusion.

Collapse. If R is not positive, we show that reductional equality collapses: \leftrightarrow identifies terms which bear absolutely no relationship with each other.

Lemma 13. *Assume, there are $a, b \in R^{\bullet}$ such that $a + b = 0$, then for all term σ , $\mathbf{0} \widetilde{\rightarrow}^* a\sigma \widetilde{\rightarrow}^* \mathbf{0}$.*

Proof. Take Y a fixed point combinator of λ -calculus, such that $(Y)s \rightarrow_{\beta}^* (s)(Y)s$ for all λ -term s . Write ∞_{σ} for $(Y)\lambda x(\sigma + x)$; then $\infty_{\sigma} \widetilde{\rightarrow}^* \sigma + \infty_{\sigma}$. We get:

$$\mathbf{0} \stackrel{\nabla}{=} a\infty_{\sigma} + b\infty_{\sigma} \widetilde{\rightarrow}^* a\sigma + a\infty_{\sigma} + b\infty_{\sigma} \stackrel{\nabla}{=} a\sigma$$

and

$$a\sigma \stackrel{\nabla}{=} a\sigma + a\infty_{\sigma} + b\infty_{\sigma} \widetilde{\rightarrow}^* a\sigma + a\infty_{\sigma} + b\sigma + b\infty_{\sigma} \stackrel{\nabla}{=} \mathbf{0}.$$

As an immediate corollary, \leftrightarrow identifies any two terms as soon as 1 has an opposite in R .

Corollary 3. *If R is such that 1 has an opposite, i.e. $-1 \in R$ with $1 + (-1) = 0$, then for all terms σ and τ , $\sigma \widetilde{\rightarrow}^* \tau$.*

4 On Normalization

Unsurprisingly, if R is not positive, there is no normal term: assume there are $a, b \in R$ such that $a + b = 0$ and $a \neq 0$ and let $s \in A_R$ and $S' \in C_R$ be such that $s \rightarrow S'$; then for all $\sigma \in R\langle \Delta_R \rangle$, $\sigma \stackrel{\nabla}{=} as + bs + \sigma$ and then $\sigma \widetilde{\rightarrow} aS' + bs + \sigma$. Hence every term σ reduces.

Moreover, even if R is positive, it may be the case that the only normalizable terms are normal terms. Indeed, assume R is the set \mathbf{Q}^+ of non-negative rational numbers (which is a positive rig) and let $s \in A_R$ and $S' \in C_R$ be such that $s \rightarrow S'$; then there is an infinite sequence of reductions from s :

$$s \stackrel{\nabla}{=} \frac{1}{2}s + \frac{1}{2}s \widetilde{\rightarrow} \frac{1}{2}s + \frac{1}{2}S' \widetilde{\rightarrow} \frac{1}{4}s + \frac{3}{4}S' \widetilde{\rightarrow} \dots \widetilde{\rightarrow} \frac{1}{2^n}s + \frac{2^n - 1}{2^n}S' \widetilde{\rightarrow} \dots$$

In [ER03], it is proved that if R is the set \mathbf{N} of all natural numbers, then simply typed terms are strongly normalizing. The associated type system is defined on canonical terms, by adding the following rules for linear combinations:

$$\frac{}{\Gamma \vdash \mathbf{0} : A} \qquad \frac{\Gamma \vdash \sigma : A}{\Gamma \vdash a\sigma : A} \qquad \frac{\Gamma \vdash \sigma : A \quad \Gamma \vdash \tau : A}{\Gamma \vdash \sigma + \tau : A}$$

to usual typing rules for variable, abstraction and application. Then one extends typing to all terms: for all $\sigma \in R\langle\Delta_R\rangle$, write $\Gamma \vdash \sigma : A$ iff $\Gamma \vdash \text{can}(\sigma) : A$. The strong normalization proof is by an adaptation of Tait’s reducibility method, as presented in [Kri90], using the following key lemma:

Lemma 14. *The set of all strongly normalizing terms is the R -submodule of $R\langle\Delta_R\rangle$ generated by simple strongly normalizing terms: i.e. σ is strongly normalizing iff, for all $s \in \text{Supp}(\sigma)$, s is strongly normalizing.*

which is easily established in the case $R = \mathbf{N}$.

In this section, we sketch a proof of strong normalization in a more general case. A thorough development on normalization, including a full proof of Theorem 3 is provided in [Vau06]. In [Vau07], the author showed that Lemma 14 can be generalized to any rig R such that:

- (i) R is finitely splitting in the sense that, for all $a \in R$, a has finitely many writings as a sum: $\{(a_1, \dots, a_n) \in (R^\bullet)^n; n \in \mathbf{N} \text{ and } a = a_1 + \dots + a_n\}$ is always finite;
- (ii) the width function $w : R \rightarrow \mathbf{N}$ defined by

$$w(a) = \max \{n \in \mathbf{N}; \exists (a_1, \dots, a_n) \in (R^\bullet)^n \text{ s.t. } a = a_1 + \dots + a_n\}$$

is a morphism of rigs: $w(a + b) = w(a) + w(b)$ and $w(ab) = w(a)w(b)$.

Clearly, these conditions also imply R is positive.

Example 1. Setting $R = \mathbf{N}$ satisfies these conditions, with $w(n) = n$ for all $n \in \mathbf{N}$. One more interesting instance is the rig $\mathbf{N}[\xi_1, \dots, \xi_n]$ of all polynomials over indeterminates ξ_1, \dots, ξ_n with non-negative integer coefficients: the width of polynomial P is its value at point $(1, \dots, 1)$, i.e. the sum of its coefficients. Conversely, if conditions (ii) and (iii) hold in R , then all elements of R are sums of elements of width 1.

Condition (ii) entails that every term has finitely many \rightsquigarrow -reducts. Hence, for all strongly normalizing term σ , König’s lemma implies that the length of a sequence of \rightsquigarrow -reductions from σ is bounded. One then proves Lemma 14 by induction on the maximal length of a sequence of reductions from σ : in this proof, condition (iii) is crucial to enable the use of the inductive hypothesis.

One can then generalize the proof of strong normalization from [ER03] to this setting. The reducibility method amounts to consider saturated sets of terms, which are closed under backwards head linear reduction. Then one proves that the set \mathcal{N} of all strongly normalizing terms is saturated: this involves Lemma 14. Last, one naturally interprets simple propositional types by saturated subsets of \mathcal{N} , and prove that typable terms lie in the interpretation of their types. As a corollary, we obtain:

Theorem 3. *If R is finitely splitting and w is a morphism of rigs, then all typable terms are strongly normalizing.*

Theorem 3 may be used to provide a weak normalization result in a more general case:

Corollary 4. *If R is positive, then every typable term admits a normal form.*

The algorithm behind Corollary 4 can be sketched as follows, assuming the term $\sigma \in R\langle\Delta_R\rangle$ is typable:

- replace scalars occurring in $\text{can}(\sigma)$ with formal indeterminates;
- the object τ thus obtained can be considered as a term with coefficients in the free rig generated by indeterminates;
- this rig of polynomials enjoys (ii) and (iii), hence Theorem 3 applies, since τ is also typable;
- replace indeterminates by their values in the normal form of τ : this is the normal form of σ .

This technique may also be used to reduce terms while temporarily disabling interaction between reduction and coefficients: replace coefficients in the canonical form with indeterminates, reduce, then evaluate polynomials. This, however, does not provide a suitable notion of reduction, since it is not confluent (it is very similar to reduction $\widehat{\rightarrow}$ outlined in next section).

5 Other Approaches and Future Work

It is noteworthy that the collapse we described in section 3.3 involves a term ∞_σ such that $\infty_\sigma \xrightarrow{*} n\sigma + \infty_\sigma$, for all $n \in \mathbf{N}$: reduction of ∞_σ generates a potentially infinite amount of σ . This is not a surprise, since untyped algebraic λ -calculus involves both linear algebra and arbitrary fixed points. The *raw* term $\infty_\sigma + (-1)\infty_\sigma$ is then analogous to the well know indeterminate form $\infty - \infty$ of the affinely extended real line. The collapse of reduction in presence of negative scalars follows from the fact that we consider $\mathbf{0} \stackrel{\forall}{=} \infty_\sigma - \infty_\sigma$.

Also, we have seen that the way we defined reduction is problematic w.r.t. normalization properties: even if R is positive, typable terms needn't be strongly normalizing. Here we briefly review some possible other approaches.

Restricting Reduction. One seemingly natural variant of one-step reduction is the following one. Rather than (3), extend reduction from simple terms to all terms by:

$$\sigma \widehat{\rightarrow} \sigma' \text{ if } \sigma \stackrel{\forall}{=} as + T \text{ and } \sigma' \stackrel{\forall}{=} aS' + T, \text{ with } a \neq 0, T_{(s)} = 0 \text{ and } s \rightarrow S' . \quad (5)$$

This amounts to restrict the contextuality of reduction to the canonical forms of terms. This reduction, however, is not confluent: $\infty_y + (\lambda x x) \infty_y \widehat{\rightarrow}^*$ -reduces to both $2\infty_y$ and $y + 2\infty_y$, and these have no common $\widehat{\rightarrow}^*$ -reduct (many thanks to the referee who provided this very simple argument). Nonetheless, $\widehat{\rightarrow}$ is trivially conservative over ordinary β -reduction: the $\widehat{\rightarrow}$ -reducts of a λ -term are exactly its β -reducts. Also, $\widehat{\rightarrow}$ should be well-behaved as far as normalization is concerned: the trick involving rational coefficients is no longer possible.

Typing. The restriction we have just suggested diminishes the role of scalar operations during reduction. Another possible fix to the collapse might involve typing, in order to ward arbitrary fixed points off. The main problem with that idea is that, unless the set of scalars is positive, typing is not preserved by reduction. Hence, it seems better to study the denotational semantics of ordinary typed λ -calculus in finiteness spaces [Ehr05] more thoroughly, before investigating further in that direction.

Restricting Equality. Last, we mention a completely different point of view on linear combinations of terms. In [AD06], Arrighi and Dowek introduce linear algebraic λ -calculus. The background setting is quite unrelated: their work provides a framework for quantum computation; in particular, terms represent linear operators, hence application is bilinear rather than linear in the function only. Notwithstanding this distinction, their approach to λ -calculus with linear combinations of terms contrasts with ours: consider terms up to \equiv rather than some variant of $\stackrel{\nabla}{=}$, and handle the identities between linear combinations, together with analogues of (II), as reduction rules.

Confronted to problems similar to those we exposed above in presence of negative coefficients, they opted for a completely different solution, far more natural in their setting: restrict those reduction rules involving rewriting of linear combinations to closed terms in normal form. This allows to tame some of the intrinsic potential infinity of pure λ -calculus, avoiding to consider indeterminate forms. Up to these restrictions, they prove confluence for the whole system.

It will be particularly interesting to find out whether a similar system can be successfully defined in the setting of algebraic λ -calculus.

References

- [AD05] Arrighi, P., Dowek, G.: A computational definition of the notion of vectorial space. *Electr. Notes Theor. Comput. Sci.* 117, 249–261 (2005)
- [AD06] Arrighi, P., Dowek, G.: Linear-algebraic lambda-calculus: higher-order, encodings and confluence (2006) Manuscript Available at <http://www.lix.polytechnique.fr/~dowek/publi.html>
- [Coq] The Coq proof assistant <http://coq.inria.fr/>
- [Ehr05] Ehrhard, T.: Finiteness spaces. *Mathematical Structures in Comp. Sci.* 15(4), 615–646 (2005)
- [ER03] Ehrhard, T., Regnier, L.: The differential lambda-calculus. *Theoretical Computer Science* 309, 1–41 (2003)
- [Kri90] Krivine, J.-L.: *Lambda-calcul, types et modèles*. Masson, Paris (1990)
- [PS81] Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. *J. ACM* 28(2), 233–264 (1981)
- [Vau06] Vaux, L.: λ -calculus in an algebraic setting. (2006) Research report available at <http://iml.univ-mrs.fr/~vaux/articles/alglam.ps.gz>
- [Vau07] Vaux, L.: The differential $\lambda\mu$ -calculus. To appear in *Theoretical Computer Science* (2007)

Satisfying KBO Constraints^{*}

Harald Zankl and Aart Middeldorp

Institute of Computer Science
University of Innsbruck
Austria

Abstract. This paper presents two new approaches to prove termination of rewrite systems with the Knuth-Bendix order efficiently. The constraints for the weight function and for the precedence are encoded in (pseudo-)propositional logic and the resulting formula is tested for satisfiability. Any satisfying assignment represents a weight function and a precedence such that the induced Knuth-Bendix order orients the rules of the encoded rewrite system from left to right.

1 Introduction

This paper is concerned with proving termination of term rewrite systems (TRSs) with the Knuth-Bendix order (KBO), a method invented by Knuth and Bendix in [14] well before termination research in term rewriting became a very popular and competitive endeavor (as witnessed by the annual termination competition)¹. We know of only two termination tools that contain an implementation of KBO, AProVE [11] and TTT [12], but neither of these tools incorporate KBO in their fully automatic mode for the TRS category. This is perhaps due to the fact that the algorithms known for deciding KBO orientability ([5,15]) are not easy to implement efficiently, despite the fact that the problem is known to be decidable in polynomial time [15]. The aim of this paper is to make KBO a more attractive choice for termination tools by presenting two simple encodings of KBO orientability into (pseudo-)propositional logic such that checking satisfiability of the resulting formula amounts to proving KBO termination.

Kurihara and Kondo [16] were the first to encode a termination method for term rewriting into propositional logic. They showed how to encode orientability with respect to the lexicographic path order as a satisfaction problem. Codish *et al.* [3] presented a more efficient formulation for the properties of a precedence. In [4,22] encodings of argument filterings are presented which can be combined with propositional encodings of reduction pairs in order to obtain logic-based implementations of the dependency pair method. Propositional encodings of other termination methods are described in [9,10,13].

In Section 2 the necessary definitions for KBO are presented. Section 3 introduces a purely propositional encoding of KBO also describing the optimizations

^{*} This research is supported by FWF (Austrian Science Fund) project P18763. Some of the results in this paper were first announced in [23].

¹ www.lri.fr/~marche/termination-competition

applied in the implementation. In Section 4 an alternative encoding is given using pseudo-boolean constraints. We compare the power and run times of our implementations with the ones of AProVE and TTT in Section 5 and show the enormous gain in efficiency. We draw some conclusions in Section 6. One of these is that our pseudo-boolean encoding of KBO revealed a bug in MiniSat+. Section 7 summarizes the main contributions of this paper.

2 Preliminaries

We assume familiarity with the basics of term rewriting (e.g. [2]). In this preliminary section we recall the definition of KBO. A *quasi-precedence* \succsim (*strict precedence* \succ) is a quasi-order (proper order) on a signature \mathcal{F} . Sometimes we find it convenient to call a quasi-precedence simply precedence. A *weight function* for a signature \mathcal{F} is a pair (w, w_0) consisting of a mapping $w: \mathcal{F} \rightarrow \mathbb{N}$ and a constant $w_0 > 0$ such that $w(c) \geq w_0$ for every constant $c \in \mathcal{F}$. Let \mathcal{F} be a signature and (w, w_0) a weight function for \mathcal{F} . The *weight* of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is defined as follows:

$$w(t) = \begin{cases} w_0 & \text{if } t \text{ is a variable,} \\ w(f) + \sum_{i=1}^n w(t_i) & \text{if } t = f(t_1, \dots, t_n). \end{cases}$$

A weight function (w, w_0) is *admissible* for a quasi-precedence \succsim if $f \succsim g$ for all function symbols g whenever f is a unary function symbol with $w(f) = 0$. For a term t , $|t|$ denotes its size and $|t|_a$ for $a \in \mathcal{F} \cup \mathcal{V}$ denotes how often the symbol a occurs in t .

Definition 1 ([14,5,19]). *Let \succsim be a quasi-precedence and (w, w_0) a weight function. We define the Knuth-Bendix order $>_{\text{kbo}}$ on terms inductively as follows: $s >_{\text{kbo}} t$ if $|s|_x \geq |t|_x$ for all variables $x \in \mathcal{V}$ and either*

- (a) $w(s) > w(t)$, or
- (b) $w(s) = w(t)$ and one of the following alternatives holds:
 - (1) $t \in \mathcal{V}$, $s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\})$, and $s \neq t$, or
 - (2) $s = f(s_1, \dots, s_n)$, $t = g(t_1, \dots, t_m)$, $f \sim g$, and there exists an $1 \leq i \leq \min\{n, m\}$ such that $s_i >_{\text{kbo}} t_i$ and $s_j = t_j$ for all $1 \leq j < i$, or
 - (3) $s = f(s_1, \dots, s_n)$, $t = g(t_1, \dots, t_m)$, and $f \succ g$.

where $\mathcal{F}^{(n)}$ denotes the set of all function symbols $f \in \mathcal{F}$ of arity n . Thus in case (b)(1) the term s consists of a nonempty sequence of unary function symbols applied to the variable t .

Specializing the above definition to (the reflexive closure of) a strict precedence, one obtains the definition of KBO in [2], except that we restrict weight functions to have range \mathbb{N} instead of \mathbb{R} . According to [15] this does not decrease the power of the order.

Lemma 2. *A TRS \mathcal{R} is terminating whenever there exist a quasi-precedence \succsim and a weight function (w, w_0) such that $\mathcal{R} \subseteq \succ_{\text{kbo}}$. \square*

Example 3. The TRS SK_90.2.42² consisting of the rules

$$\begin{array}{ll}
 \text{flatten}(\text{nil}) \rightarrow \text{nil} & \text{rev}(\text{nil}) \rightarrow \text{nil} \\
 \text{flatten}(\text{unit}(x)) \rightarrow \text{flatten}(x) & \text{rev}(\text{unit}(x)) \rightarrow \text{unit}(x) \\
 \text{flatten}(x ++ y) \rightarrow \text{flatten}(x) ++ \text{flatten}(y) & \text{rev}(x ++ y) \rightarrow \text{rev}(y) ++ \text{rev}(x) \\
 \text{flatten}(\text{unit}(x) ++ y) \rightarrow \text{flatten}(x) ++ \text{flatten}(y) & \text{rev}(\text{rev}(x)) \rightarrow x \\
 \text{flatten}(\text{flatten}(x)) \rightarrow \text{flatten}(x) & (x ++ y) ++ z \rightarrow x ++ (y ++ z) \\
 x ++ \text{nil} \rightarrow x & \text{nil} ++ y \rightarrow y
 \end{array}$$

is KBO terminating. The weight function (w, w_0) with $w(\text{flatten}) = w(\text{rev}) = w(++) = 0$ and $w(\text{unit}) = w(\text{nil}) = w_0 = 1$ together with the quasi-precedence $\text{flatten} \sim \text{rev} \succ \text{unit} \succ ++ \succ \text{nil}$ ensures that $l \succ_{\text{kbo}} r$ for all rules $l \rightarrow r$. The use of a quasi-precedence is essential here; the rules $\text{flatten}(x ++ y) \rightarrow \text{flatten}(x) ++ \text{flatten}(y)$ and $\text{rev}(x ++ y) \rightarrow \text{rev}(y) ++ \text{rev}(x)$ demand $w(\text{flatten}) = w(\text{rev}) = 0$ but KBO with strict precedence does not allow different unary functions to have weight zero.

One can imagine a more general definition of KBO. For instance, in case (b)(2) we could demand that $s_j \sim_{\text{kbo}} t_j$ for all $1 \leq j < i$ where $s \sim_{\text{kbo}} t$ if and only if $s \sim t$ and $w(s) = w(t)$. Here $s \sim t$ denotes syntactic equality with respect to equivalent function symbols of the same arity. Another obvious extension would be to compare the arguments according to an arbitrary permutation or as multisets. To keep the discussion and implementation simple, we do not consider such refinements in the sequel.

3 A Pure SAT Encoding of KBO

In order to give a propositional encoding of KBO termination, we must take care of representing a precedence and a weight function. For the former we introduce two sets of new variables $X = \{X_{fg} \mid f, g \in \mathcal{F} \text{ with } f \neq g\}$ and $Y = \{Y_{fg} \mid f, g \in \mathcal{F} \text{ with } f \neq g\}$ depending on the underlying signature \mathcal{F} ([16,21]). The intended semantics of these variables is that an assignment which satisfies a variable X_{fg} corresponds to a precedence with $f \succ g$ and similarly Y_{fg} suggests $f \sim g$. When dealing with strict precedences it is safe to assign all Y_{fg} variables to false. For the weight function, symbols are considered in binary representation and the operations $>$, $=$, \geq , and $+$ must be redefined accordingly. The propositional encodings of $>$ and $=$ given below are similar to the ones in [3]. To save parentheses we employ the binding hierarchy for the connectives where $+$ binds strongest, followed by the relation symbols $>$, $=$, and \geq . The logical connectives \vee and \wedge are next in the hierarchy and \rightarrow and \leftrightarrow bind weakest.

² Labels in sans-serif font refer to TRSs in the Termination Problems Data Base [18].

We fix the number k of bits that is available for representing natural numbers in binary. Let $a < 2^k$. We denote by $\mathbf{a} = \langle a_k, \dots, a_1 \rangle$ the binary representation of a where a_k is the most significant bit.

Definition 4. For natural numbers given in binary representation, the operations $>$, $=$, and \geq are defined as follows (for all $1 \leq j \leq k$):

$$\begin{aligned} \mathbf{f} >_j \mathbf{g} &= \begin{cases} f_1 \wedge \neg g_1 & \text{if } j = 1 \\ (f_j \wedge \neg g_j) \vee ((f_j \leftrightarrow g_j) \wedge \mathbf{f} >_{j-1} \mathbf{g}) & \text{if } j > 1 \end{cases} \\ \mathbf{f} > \mathbf{g} &= \mathbf{f} >_k \mathbf{g} \\ \mathbf{f} = \mathbf{g} &= \bigwedge_{i=1}^k (f_i \leftrightarrow g_i) \\ \mathbf{f} \geq \mathbf{g} &= \mathbf{f} > \mathbf{g} \vee \mathbf{f} = \mathbf{g} \end{aligned}$$

Next we define a formula which is satisfiable if and only if the encoded weight function is admissible for the encoded precedence.

Definition 5. For a weight function (w, w_0) , let $\text{ADM-SAT}(w, w_0)$ be the formula

$$w_0 > \mathbf{0} \wedge \bigwedge_{c \in \mathcal{F}^{(0)}} \mathbf{c} \geq w_0 \wedge \bigwedge_{f \in \mathcal{F}^{(1)}} (\mathbf{f} = \mathbf{0} \rightarrow \bigwedge_{g \in \mathcal{F}, f \neq g} (X_{fg} \vee Y_{fg}))$$

For addition we use pairs. The first component represents the bit representation and the second component is a propositional formula which encodes the constraints for each digit.

Definition 6. We define $(\mathbf{f}, \varphi) + (\mathbf{g}, \psi)$ as $(\mathbf{s}, \varphi \wedge \psi \wedge \gamma \wedge \sigma)$ with

$$\gamma = \neg c_k \wedge \neg c_0 \wedge \bigwedge_{i=1}^k (c_i \leftrightarrow ((f_i \wedge g_i) \vee (f_i \wedge c_{i-1}) \vee (g_i \wedge c_{i-1})))$$

and

$$\sigma = \bigwedge_{i=1}^k (s_i \leftrightarrow (f_i \oplus g_i \oplus c_{i-1}))$$

where c_i ($0 \leq i \leq k$) and s_i ($1 \leq i \leq k$) are fresh variables that represent the carry and the sum of the addition and \oplus denotes exclusive or. The condition $\neg c_k$ prevents a possible overflow.

Note that although theoretically not necessary, it is a good idea to introduce new variables for the sum. The reason is that in consecutive additions each bit f_i and g_i is duplicated (twice for the carry and once for the sum) and consequently using fresh variables for the sum prevents an exponential blowup of the resulting formula.

Definition 7. We define $(\mathbf{f}, \varphi) > (\mathbf{g}, \psi)$ as $\mathbf{f} > \mathbf{g} \wedge \varphi \wedge \psi$ and $(\mathbf{f}, \varphi) = (\mathbf{g}, \psi)$ as $\mathbf{f} = \mathbf{g} \wedge \varphi \wedge \psi$.

In the next definition we show how the weight of terms is computed propositionally.

Definition 8. Let t be a term and (w, w_0) a weight function. The weight of a term is encoded as follows:

$$W_t = \begin{cases} (w_0, \top) & \text{if } t \in \mathcal{V}, \\ (\mathbf{f}, \top) + \sum_{i=1}^n W_{t_i} & \text{if } t = f(t_1, \dots, t_n). \end{cases}$$

We are now ready to define a propositional formula that reflects the definition of $>_{\text{kbo}}$.

Definition 9. Let s and t be terms. We define the formula $\text{SAT}(s >_{\text{kbo}} t)$ as follows. If $s \in \mathcal{V}$ or $s = t$ or $|s|_x < |t|_x$ for some $x \in \mathcal{V}$ then $\text{SAT}(s >_{\text{kbo}} t) = \perp$. Otherwise

$$\text{SAT}(s >_{\text{kbo}} t) = W_s > W_t \vee (W_s = W_t \wedge \text{SAT}(s >'_{\text{kbo}} t))$$

with

$$\text{SAT}(s >'_{\text{kbo}} t) = \begin{cases} \top & \text{if } t \in \mathcal{V}, s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\}), \text{ and } s \neq t \\ \text{SAT}(s_i >_{\text{kbo}} t_i) & \text{if } s = f(s_1, \dots, s_n), t = f(t_1, \dots, t_n) \\ X_{fg} \vee (Y_{fg} \wedge \text{SAT}(s_i >_{\text{kbo}} t_i)) & \text{if } s = f(s_1, \dots, s_n), t = g(t_1, \dots, t_m), \text{ and } f \neq g \end{cases}$$

where in the second (third) clause i denotes the least $1 \leq j \leq n$ ($\min\{n, m\}$) with $s_j \neq t_j$.

3.1 Encoding the Precedence in SAT

To ensure the properties of a precedence we follow the approach of Codish et al. [3] who propose to interpret function symbols as natural numbers. The greater than or equal to relation then ensures that the function symbols are quasi-ordered. Let $|\mathcal{F}| = n$. We are looking for a mapping $m: \mathcal{F} \rightarrow \{1, \dots, n\}$ such that for every propositional variable $X_{fg} \in X$ we have $m(f) > m(g)$ and for $Y_{fg} \in Y$ we get $m(f) = m(g)$. To uniquely encode one of the n function symbols, $l := \lceil \log_2(n) \rceil$ fresh propositional variables are needed. The l -bit representation of f is $\langle f'_1, \dots, f'_l \rangle$ with f'_l the most significant bit.

Definition 10. For all $1 \leq j \leq l$

$$\|X_{fg}\|_j = \begin{cases} f'_1 \wedge \neg g'_1 & \text{if } j = 1 \\ (f'_j \wedge \neg g'_j) \vee ((f'_j \leftrightarrow g'_j) \wedge \|X_{fg}\|_{j-1}) & \text{if } j > 1 \end{cases}$$

$$\|Y_{fg}\|_l = \bigwedge_{j=1}^l (f'_j \leftrightarrow g'_j)$$

Note that the variables f'_i ($1 \leq i \leq l$) are different from f_i ($1 \leq i \leq k$) which are used to represent weights.

Definition 11. Let \mathcal{R} be a TRS. The formula $\text{KBO-SAT}(\mathcal{R})$ is defined as

$$\text{ADM-SAT}(w, w_0) \wedge \bigwedge_{l \rightarrow r \in \mathcal{R}} \text{SAT}(l \succ_{\text{kbo}} r) \wedge \bigwedge_{z \in X \cup Y} (z \leftrightarrow ||z||_l)$$

Theorem 12. A TRS \mathcal{R} is terminating whenever the propositional formula $\text{KBO-SAT}(\mathcal{R})$ is satisfiable. \square

The reverse does not hold (Example [21](#)).

3.2 Optimizations

This section deals with logical simplifications concerning propositional formulas as well as optimizations which are specific for the generation of the constraint formula which encodes KBO termination of the given instance.

Logical Optimizations. Since the constraint formula contains many occurrences of \top and \perp logical equivalences simplifying such formulas are employed.

SAT solvers typically expect their input in conjunctive normal form (CNF) but for the majority of the TRSs the constraint formula $\text{KBO-SAT}(\mathcal{R})$ is too large for the standard translation. The problem is that the resulting CNF may be exponentially larger than the input formula because when distributing \vee over \wedge subformulas get duplicated. In [20](#) Tseitin proposed a transformation which is linear in the size of the input formula. The price for linearity is paid with introducing new variables. As a consequence, Tseitin's transformation does not produce an equivalent formula, but it does preserve and reflect satisfiability.

Optimizations Concerning the Encoding. Before discussing the implemented optimizations in detail it is worth mentioning the bottleneck of the whole procedure. As addressed in the previous section, SAT solvers expect their input in CNF. It turned out that the generation of all non-atomic subformulas, which are needed for the translation, constitutes the main bottleneck. So every change in the implementation which reduces the size of the constraint formula will result in an additional speedup. All improvements discussed in the sequel could reduce the execution time at least a bit. Whenever they are essential it is explicitly stated.

Since \succ_{kbo} is a simplification order it contains the embedding relation. We make use of that fact by only computing the constraint formula $s \succ_{\text{kbo}} t$ if the test $s \triangleright_{\text{emb}} t$ is false. Most of the other optimizations deal with representing or computing the weight function. When computing the constraints for the weights in a rule $l \rightarrow r$, removing function symbols and variables that occur both in l and in r is highly recommended or even necessary for an efficient implementation. The benefit can be seen in the example below. Note that propositional addition is somehow expensive as new variables have to be added for representing the

carry and the sum in addition to a formula which encodes the constraints for each digit.

Example 13. Consider the TRS consisting of the single rule $f(y, g(x), x) \rightarrow f(y, x, g(g(x)))$. Without the optimization the constraints for the weights would amount to

$$\begin{aligned} (\mathbf{f}, \top) + (\mathbf{w}_0, \top) + (\mathbf{g}, \top) + (\mathbf{w}_0, \top) + (\mathbf{w}_0, \top) \\ \geq \\ (\mathbf{f}, \top) + (\mathbf{w}_0, \top) + (\mathbf{w}_0, \top) + (\mathbf{g}, \top) + (\mathbf{g}, \top) + (\mathbf{w}_0, \top) \end{aligned}$$

whereas employing the optimization produces the more or less trivial constraint $(\mathbf{0}, \top) \geq (\mathbf{g}, \top)$.

By using a cache for propositional addition we can test if we already computed the sum of the weights of two function symbols f and g . That reduces the number of newly introduced variables and sometimes we can omit the constraint formula for addition. This is clarified in the following example.

Example 14. Consider the TRS consisting of the rules $f(\mathbf{a}) \rightarrow \mathbf{b}$ and $f(\mathbf{a}) \rightarrow \mathbf{c}$. The constraints for the first rule amount to the following formula where $\underline{\mathbf{fa}}$ corresponds to the new variables which are required for the sum when adding f and \mathbf{a} and the propositional formula φ represents the constraints which are put on each digit of $\underline{\mathbf{fa}}$:

$$\begin{aligned} \text{SAT}(f(\mathbf{a}) >_{\text{kbo}} \mathbf{b}) &= W_{f(\mathbf{a})} > W_{\mathbf{b}} \vee (W_{f(\mathbf{a})} = W_{\mathbf{b}} \wedge X_{f\mathbf{b}}) \\ &= (\mathbf{f}, \top) + (\mathbf{a}, \top) > (\mathbf{b}, \top) \vee ((\mathbf{f}, \top) + (\mathbf{a}, \top) = (\mathbf{b}, \top) \wedge X_{f\mathbf{b}}) \\ &= (\underline{\mathbf{fa}}, \varphi) > (\mathbf{b}, \top) \vee ((\underline{\mathbf{fa}}, \varphi) = (\mathbf{b}, \top) \wedge X_{f\mathbf{b}}) \\ &= (\underline{\mathbf{fa}} > \mathbf{b} \wedge \varphi) \vee (\underline{\mathbf{fa}} = \mathbf{b} \wedge \varphi \wedge X_{f\mathbf{b}}) \end{aligned}$$

We get a similar formula for the second rule and the conjunction of both amounts to

$$((\underline{\mathbf{fa}} > \mathbf{b} \wedge \varphi) \vee (\underline{\mathbf{fa}} = \mathbf{b} \wedge \varphi \wedge X_{f\mathbf{b}})) \wedge ((\underline{\mathbf{fa}} > \mathbf{c} \wedge \varphi) \vee (\underline{\mathbf{fa}} = \mathbf{c} \wedge \varphi \wedge X_{f\mathbf{c}}))$$

Using commutativity and distributivity we could obtain the equivalent formula

$$(\underline{\mathbf{fa}} > \mathbf{b} \vee (\underline{\mathbf{fa}} = \mathbf{b} \wedge X_{f\mathbf{b}})) \wedge (\underline{\mathbf{fa}} > \mathbf{c} \vee (\underline{\mathbf{fa}} = \mathbf{c} \wedge X_{f\mathbf{c}})) \wedge \varphi$$

which gives rise to fewer subformulas. Note that this simplification can easily be implemented using the information of the cache for addition.

4 A Pseudo-boolean Encoding of KBO

A *pseudo-boolean constraint* (PBC) is of the form

$$\left(\sum_{i=1}^n a_i * x_i \right) \circ m$$

where a_1, \dots, a_n, m are fixed integers, x_1, \dots, x_n boolean variables that range over $\{0, 1\}$, and $\circ \in \{\geq, =, \leq\}$. We separate PBCs that are written on a single line by semicolons. A sequence of PBCs is satisfiable if there exists an assignment which satisfies every PBC in the sequence. Since 2005 pseudo-boolean evaluation [17] is a track of the international SAT competition [3]. In the sequel we show how to encode KBO using PBCs.

Definition 15. For a weight function (w, w_0) let $\text{ADM-PBC}(w, w_0)$ be the collection of PBCs

$$\begin{aligned} & - \bar{w}_0 \geq 1 \\ & - \bar{w}(c) - \bar{w}_0 \geq 0 \text{ for all } c \in \mathcal{F}^{(0)} \\ & - (n - 1) * \bar{w}(f) + \sum_{f \neq g} (X_{fg} + Y_{fg}) \geq (n - 1) \text{ for all } f \in \mathcal{F}^{(1)} \end{aligned}$$

where $n = |\mathcal{F}|$, $\bar{w}(f) = 2^{k-1} * f_k + \dots + 2^0 * f_1$ denotes the weight of f in \mathbb{N} using k bits, and \bar{w}_0 denotes the value of w_0 .

In the definition above the first two PBCs express that w_0 is strictly larger than zero and that every unary function symbol has weight at least w_0 . Whenever the considered function symbol f has weight larger than zero the third constraint is trivially satisfied. In the case that the unary function symbol f has weight zero the constraints on the precedence add up to $n - 1$ if and only if f is a maximal element. Note that X_{fg} and Y_{fg} are mutual exclusive (which is ensured when encoding the constraints on a quasi-precedence, cf. Definition [18]).

For the encoding of $s >_{\text{kbo}} t$ and $s >'_{\text{kbo}} t$ auxiliary propositional variables $KBO_{s,t}$ and $KBO'_{s,t}$ are introduced. The intended meaning is that if $s >_{\text{kbo}} t$ ($s >'_{\text{kbo}} t$) then $KBO_{s,t}$ ($KBO'_{s,t}$) evaluates to true under a satisfying assignment. The general idea of the encoding is very similar to the pure SAT case. As we do not know anything about weights and the precedence at the time of encoding we have to consider the cases $w(s) > w(t)$ and $w(s) = w(t)$ at the same time. That is why $KBO'_{s,t}$ and the recursive call to $\text{PBC}(s >'_{\text{kbo}} t)$ must be considered in any case.

The weight $w(t)$ of a term t is defined similarly as in Section [2] with the only difference that the weight $w(f)$ of the function symbol $f \in \mathcal{F}$ is represented in k bits as described in Definition [15].

Definition 16. Let s and t be terms. The encoding of $\text{PBC}(s >_{\text{kbo}} t)$ amounts to $KBO_{s,t} = 0$ if $s \in \mathcal{V}$ or $s = t$ or $|s|_x < |t|_x$ for some $x \in \mathcal{V}$. In all other cases $\text{PBC}(s >_{\text{kbo}} t)$ is

$$-(m + 1) * KBO_{s,t} + w(s) - w(t) + KBO'_{s,t} \geq -m; \text{PBC}(s >'_{\text{kbo}} t)$$

where $m = 2^k * |t|$. Here $\text{PBC}(s >'_{\text{kbo}} t)$ is the empty constraint when $t \in \mathcal{V}$, $s \in \mathcal{T}(\mathcal{F}^{(1)}, \{t\})$, and $s \neq t$. In the remaining case $s = f(s_1, \dots, s_n)$, $t = g(t_1, \dots, t_m)$, and $\text{PBC}(s >'_{\text{kbo}} t)$ is the combination of $\text{PBC}(s_i >_{\text{kbo}} t_i)$ and

³ <http://sat07.ecs.soton.ac.uk>

$$\begin{cases} -KBO'_{s,t} + KBO_{s_i,t_i} \geq 0 & \text{if } f = g \\ -2 * KBO'_{s,t} + 2 * X_{fg} + Y_{fg} + KBO_{s_i,t_i} \geq 0 & \text{if } f \neq g \end{cases}$$

where i denotes the least $1 \leq j \leq \min\{n, m\}$ with $s_i \neq t_i$.

Since the encoding of $PBC(s >_{kbo} t)$ is explained in the example below here we just explain the intended semantics of $PBC(s >'_{kbo} t)$. In the first case where t is a variable there are no constraints on the weights and the precedence which means that the empty constraint is returned. In the case where s and t have identical root symbols it is demanded that whenever $KBO'_{s,t}$ holds then also KBO_{s_i,t_i} must be satisfied before going into the recursion. In the last case s and t have different root symbols and the PBC expresses that whenever $KBO'_{s,t}$ is satisfied then either $f > g$ or both $f \sim g$ and KBO_{s_i,t_i} must hold.

To get familiar with the encoding and to see why the definitions are a bit tricky consider the example below. For reasons of readability symbols occurring both in s and in t are removed immediately. This entails that the multiplication factor m should be lowered to

$$m = \sum_{a \in \mathcal{F} \cup \mathcal{V}} \max\{0, 2^k * (|t|_a - |s|_a)\},$$

which again is a lower bound of the left-hand side of the constraint if $KBO_{s,t}$ is false.

Example 17. Consider the TRS consisting of the rule

$$s = f(g(x), g(g(x))) \rightarrow f(g(g(x)), x) = t$$

The PB encoding $PBC(s >_{kbo} t)$ then looks as follows:

$$-KBO_{s,t} + w(g) + KBO'_{s,t} \geq 0 \tag{1}$$

$$-KBO'_{s,t} + KBO_{g(x),g(g(x))} \geq 0 \tag{2}$$

$$-(2^k + 1) * KBO_{g(x),g(g(x))} - w(g) + KBO'_{g(x),g(g(x))} \geq -2^k \tag{3}$$

$$KBO'_{g(x),g(g(x))} + KBO_{x,g(x)} \geq 0 \tag{4}$$

$$KBO_{x,g(x)} = 0 \tag{5}$$

Constraint (1) states that if $s >_{kbo} t$ then either $w(g) > 0$ or $s >'_{kbo} t$. Clearly the attentive reader would assign $w(g) = 1$ and termination of the TRS is shown. The encoding however is not so smart and performs the full recursive translation to PB. In (3) it is not possible to satisfy $s_1 = g(x) >_{kbo} g(g(x)) = t_1$ since the former is embedded in the latter. Nevertheless the constraint (3) must remain satisfiable because the TRS is KBO terminating. The trick is to introduce a hidden case distinction. The multiplication factor in front of the KBO_{s_1,t_1} variable does that job. Whenever $s_1 >_{kbo} t_1$ is needed then KBO_{s_1,t_1} must evaluate to true. Then implicitly the constraint demands that $w(s_1) > w(t_1)$ or $w(s_1) = w(t_1)$ and $s_1 >'_{kbo} t_1$ which reflects the definition of KBO. If $s_1 >_{kbo} t_1$

need not be satisfied (e.g., because already $s >_{\text{kbo}} t$ in (1)) then the constraint holds in any case since the left hand side in (3) never becomes smaller than -2^k .

4.1 Encoding the Precedence in PBCs

To encode a precedence in PB we again interpret function symbols in \mathbb{N} . For this approach an additional set of propositional variables $Z = \{Z_{fg} \mid f, g \in \mathcal{F} \text{ with } f \neq g\}$ is used. The intended semantics is that Z_{fg} evaluates to true whenever $g \succ f$ or f and g are incomparable. Just note that the Z_{fg} variables are not necessary as far as termination proving power is considered but they are essential to encode partial precedences which are sometimes handy (as explained in Section 6).

Definition 18. For a signature \mathcal{F} we define $\text{PREC-PBC}(\mathcal{F})$ using the PBCs below. Let $l = \lceil \log_2(|\mathcal{F}|) \rceil$. For all $f, g \in \mathcal{F}$ with $f \neq g$

$$\begin{aligned} 2 * X_{fg} + Y_{fg} + Y_{gf} + 2 * Z_{fg} &= 2 \\ -X_{fg} + 2^l * Y_{fg} + 2^l * Z_{fg} + i(f) - i(g) &\geq 0 \\ 2^l * X_{fg} + Y_{fg} + 2^l * Z_{fg} + i(f) - i(g) &\geq 1 \end{aligned}$$

where $i(f) = 2^{l-1} * f'_l + \dots + 2^0 * f'_1$ denotes the interpretation of f in \mathbb{N} using l bits.

The above definition expresses all requirements of a quasi-precedence. The symmetry of \sim and the mutual exclusion of the X , Y , and Z variables is mimicked by the first constraint. The second constraint encodes the conditions that are put on the X variables. Whenever a system needs $f > g$ in the precedence to be terminating then X_{fg} must evaluate to true and (because they are mutually exclusive) Y_{fg} and Z_{fg} to false. Hence in order to remain satisfiable $i(f) > i(g)$ must hold. In a case where $f > g$ is not needed (but the TRS is KBO terminating) the constraint must remain satisfiable. Thus Y_{fg} or Z_{fg} evaluate to one and because $i(g)$ is bound by $2^l - 1$ the constraint does no harm. Summing up, the second constraint encodes a proper order on the symbols in \mathcal{F} . The third constraint forms an equivalence relation on \mathcal{F} using the Y_{fg} variables. Whenever $f \sim g$ is demanded somehow in the encoding, then X_{fg} and Z_{fg} evaluate to false by the first constraint. Satisfiability of the third constraint implies $i(f) \geq i(g)$ but at the same time symmetry demands that Y_{gf} also evaluates to true which leads to $i(g) \geq i(f)$ and thus to $i(f) = i(g)$.

Definition 19. Let \mathcal{R} be a TRS. The pseudo-boolean encoding $\text{KBO-PBC}(\mathcal{R})$ is defined as the combination of $\text{ADM-PBC}(w, w_0)$, $\text{PREC-PBC}(\mathcal{F})$, and

$$\text{PBC}(l >_{\text{kbo}} r); \text{KBO}_{l,r} = 1$$

for all $l \rightarrow r \in \mathcal{R}$.

Theorem 20. A TRS \mathcal{R} is terminating whenever the PBCs $\text{KBO-PBC}(\mathcal{R})$ are satisfiable. □

Again the reverse does not hold (Example 21).

5 Experimental Results

We implemented our encodings on top of T_TT [12], MiniSat and MiniSat+ [7,8] were used to check satisfiability of the SAT and PBC based encodings. Below we compare our implementations of KBO, `sat` and `pbc`, with the ones of T_TT and AProVE [11]. T_TT admits only strict precedences, AProVE also quasi-precedences. Both implement the polynomial time algorithm of Korovin and Voronkov [15] together with techniques of Dick *et al.* [5].

We used the 865 TRSs which do not specify any strategy or theory and the 322 string rewrite systems (SRSS) in version 3.2 of the Termination Problem Data Base [18]. All tests were performed on a server equipped with an Intel® Xeon™ processor running at a CPU rate of 2.40 GHz and 512 MB of system memory with a timeout of 60 seconds.

5.1 Results for TRSs

As addressed in Section 3 one has to fix the number k of bits which is used to represent natural numbers in binary representation. The actual choice is specified as argument to `sat` (`pbc`). Note that a rather small k is sufficient to handle all systems from [18] which makes Theorems [12] and [20] powerful in practice. The example below gives evidence that there does not exist a general upper bound on k .

Example 21. Consider the parametrized TRS consisting of the three rules

$$f(g(x, y)) \rightarrow g(f(x), f(y)) \quad h(x) \rightarrow f(f(x)) \quad i(x) \rightarrow h^n(x)$$

with $n = 2^k$. Since the first rule duplicates the function symbol f we must assign weight zero to it. The admissibility condition for the weight function demands that f is a maximal element in the precedence. The second rule excludes the case $h \sim f$ and demands that the weight of h is strictly larger than zero. It follows that the minimum weight of $h^n(x)$ is $n + 1 = 2^k + 1$, which at the same time is the minimum weight of $i(x)$. Thus $w(i)$ is at least 2^k which requires $k + 1$ bits.

The left part of Table 1 summarizes⁴ the results for strict precedences. Since AProVE produced seriously slower results than T_TT in the TRS category, it is not considered in Table 1. Interestingly, with $k = 4$ equally many TRSs can be proved terminating as with $k = 10$. The TRS `higher-order_AProVE_HO_ReverseLastInit` needs weight eight for the constant `init` and therefore can only be proved KBO terminating with $k \geq 4$.

Concerning the optimizations in Section 3.2, if we use the standard (exponential) transformation to CNF, the total time required increases to 2681.06 seconds, the number of successful termination proofs decreases to 69, and 45 timeouts occur (for $k = 4$). Furthermore, if we don't use a cache for adding

⁴ The experiments are described in more detail at <http://cl-informatik.uibk.ac.at/~hzankl/kbo>.

Table 1. KBO for 865 TRSs

method(#bits)	strict precedence			quasi-precedence		
	total time	#successes	#timeouts	total time	#successes	#timeouts
sat/pbc(2)	19.2/16.4	72/76	0/0	20.9/16.8	73/77	0/0
sat/pbc(3)	20.2/16.3	77/77	0/0	21.9/16.9	78/78	0/0
sat/pbc(4)	21.9/16.1	78/78	0/0	22.8/17.0	79/79	0/0
sat/pbc(10)	86.1/16.7	78/78	1/0	90.2/17.2	79/79	1/0
$\mathsf{T}\mathsf{T}$	169.5	77	1			

weights and equal symbols are not removed when the weights of left and right-hand sides of rules are compared, the number of successful termination proofs remains the same but the total time increases to 92.30 seconds and one timeout occurs.

$\mathsf{T}\mathsf{T}$ without timeout requires 4747.65 seconds and can prove KBO termination of 78 TRSs. The lion’s share is taken up by `various_21` with 4016.23 seconds for a positive result. `sat(4)` needs only 0.10 seconds for this TRS and `pbc(4)` even only 0.03 seconds. Since $\mathsf{T}\mathsf{T}$ employs the slightly stronger KBO definition of [15] it can prove one TRS (`various_27`) terminating which cannot be handled by `sat` and `pbc`. On the other hand $\mathsf{T}\mathsf{T}$ gives up on `HM_t000` which specifies addition for natural numbers in decimal notation (using 104 rewrite rules). The problem is not the timeout but at some point the algorithm detects that it will require too many resources. To prevent a likely stack overflow from occurring, the computation is terminated and a “don’t know” result is reported. (AProVE behaves in a similar fashion on this TRS.) Also for our approaches this system is the most challenging one with 0.54 (`sat(4)`) and 0.11 (`pbc(4)`) seconds.

As can be seen from the right part of Table 1, by admitting quasi-precedences one additional TRS (`SK_90.2.42`, Example 3) can be proved KBO terminating. Surprisingly, AProVE 1.2 cannot prove (quasi) KBO termination of this system, for unknown reasons.

5.2 Results for SRSs

For SRSs we have similar results, as can be inferred from Table 2. The main difference is the larger number of bits needed for the propositional addition of the weights. The maximum number of SRSs is proved KBO terminating with $k \geq 7$ in case of `sat` and $k \geq 6$ for `pbc`. The reason is that in the first implementation the number of bits does not increase for intermediate sums when adding the weights. Generally speaking $\mathsf{T}\mathsf{T}$ performs better on SRSs than on TRSs concerning KBO because it can handle all systems within 546.43 seconds. The instance which consumes the most time is `Zantema_z112` with 449.01 seconds for a positive answer; `sat(7)` needs just 0.11 and `pbc(7)` 0.03 seconds. With a timeout of 60 seconds $\mathsf{T}\mathsf{T}$ proves KBO termination of 29 SRSs, without any timeout one more. Our implementations both prove KBO termination of 33 SRSs. The three SRSs that make up the difference (`Trafo_dup11`, `Zantema_z069`, `Zantema_z070`)

Table 2. KBO for 322 SRSs

method(#bits)	strict precedence			quasi-precedence		
	total time	#successes	#timeouts	total time	#successes	#timeouts
sat/pbc(2)	9.1/5.9	8/19	0/0	13.9/6.2	8/19	0/0
sat/pbc(3)	12.1/5.9	17/24	0/0	16.9/6.4	17/24	0/0
sat/pbc(4)	15.1/6.0	24/30	0/0	20.0/6.5	24/30	0/0
sat/pbc(6)	15.8/6.1	31/33	0/0	27.4/6.7	31/33	0/0
sat/pbc(7)	17.0/6.1	33/33	0/0	31.2/6.7	33/33	0/0
sat/pbc(10)	21.6/6.3	33/33	0/0	98.8/6.9	32/33	1/0
T _T T	72.4	29	1			

derive from algebra (polyhedral groups). T_TT and AProVE give up on these SRSs for the same reasons as mentioned in the preceding subsection for HM_t000.

Admitting quasi-precedences does not allow to prove KBO termination of more SRSs. On the contrary, a timeout occurs when using sat(10) on Trafo_dup11 whereas pbc(10) easily handles the system.

6 Assessment

In this section we compare the two approaches presented in this paper. Let us start with the most important measurements: power and run time. Here pbc is the clear winner. Not only is it faster on any kind of precedence; it also scales much better for larger numbers of bits used to represent the weights. Furthermore, the pseudo-boolean approach is less implementation work since additions are performed by the SAT solver and also the transformation to CNF is not necessary. We note that the implementation of pbc is exactly as described in the paper whereas sat integrates the optimizations described in Section 3.2.

A further advantage of the pseudo-boolean approach is the option of a *goal function* which should be minimized while preserving satisfiability of the constraints. Although the usage of such a goal function is not of computational interest it is useful for generating easily human readable proofs. We experimented with functions minimizing the weights for function symbols and reducing the comparisons in the precedence. The former has the advantage that one obtains a KBO proof with minimal weights which is nicely illustrated on the SRS Zan-tema_z113 consisting of the rules

$$\begin{array}{lll}
 11 \rightarrow 43 & 33 \rightarrow 56 & 55 \rightarrow 62 \\
 12 \rightarrow 21 & 22 \rightarrow 111 & 34 \rightarrow 11 \\
 44 \rightarrow 3 & 56 \rightarrow 12 & 66 \rightarrow 21
 \end{array}$$

T_TT and AProVE produce the proof

$$\begin{array}{lll}
 w(1) = 32471712256 & w(2) = 48725750528 & w(3) = 43247130624 \\
 w(4) = 21696293888 & w(5) = 44731872512 & w(6) = 40598731520 \\
 3 > 1 > 2 & & 1 > 4
 \end{array}$$

whereas `pbcb(6)` produces

$$\begin{array}{lll} w(1) = 31 & w(2) = 47 & w(3) = 41 \\ w(4) = 21 & w(5) = 43 & w(6) = 39 \\ 3 > 1 > 2 & 3 > 5 > 6 > 2 & 1 > 4 \end{array}$$

Regarding the goal function dealing with the minimization of comparisons in the precedence we detected that using two (three, four five, ten) bits to encode weights of function symbols 39 (45, 46, 47, 47) TRSs can be proved terminating in 16.7 (16.8, 17.0, 16.7, 16.9) seconds with empty precedence.

While running the experiments, `sat` and `pbcb` produced different answers for the SRS `Zantema_z13`; `pbcb` claimed KBO termination whereas `sat` answered “don’t know”. Chasing that discrepancy revealed a bug [6] in MiniSat+ (which has been corrected in the meantime).

An interesting (and probably computationally fast) extension will be the integration of the pseudo-boolean encoding of KBO into a dependency pair [1] setting [4,22]. Clearly the constraints will get more involved but we expect that the generalization to non-linear constraints in the input format for the PB track of the SAT 2007 competition will ease the work considerably.

7 Summary

In this paper we presented two logic-based encodings of KBO—pure SAT and PBC—which can be implemented more efficiently and with considerably less effort than the methods described in [5,15]. Especially the PBC encoding gives rise to a very fast implementation even without caring about possible optimizations in the encoding.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. *Theoretical Computer Science* 236, 133–178 (2000)
2. Baader, F., Nipkow, T.: *Term Rewriting and All That*. Cambridge University Press, Cambridge (1998)
3. Codish, M., Lagoon, V., Stuckey, P.: Solving partial order constraints for LPO termination. In: Pfenning, F. (ed.) *RTA 2006*. LNCS, vol. 4098, pp. 4–18. Springer, Heidelberg (2006)
4. Codish, M., Schneider-Kamp, P., Lagoon, V., Thiemann, R., Giesl, J.: SAT solving for argument filterings. In: Hermann, M., Voronkov, A. (eds.) *LPAR 2006*. LNCS (LNAI), vol. 4246, pp. 30–44. Springer, Heidelberg (2006)
5. Dick, J., Kalmus, J., Martin, U.: Automating the Knuth-Bendix ordering. *Acta Infomatica* 28, 95–119 (1990)
6. Eén, N.: Personal conversation, Google Group on Minisat (2007)
7. 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)

8. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1–26 (2006)
9. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006*. LNCS (LNAI), vol. 4130, pp. 574–588. Springer, Heidelberg (2006)
10. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: *Proc. 10th International Conference on Theory and Applications of Satisfiability Testing*. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
11. 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)
12. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool. In: Giesl, J. (ed.) *RTA 2005*. LNCS, vol. 3467, pp. 175–184. Springer, Heidelberg (2005)
13. Hofbauer, D., Waldmann, J.: Termination of string rewriting with matrix interpretations. In: Pfenning, F. (ed.) *RTA 2006*. LNCS, vol. 4098, pp. 328–342. Springer, Heidelberg (2006)
14. 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, New York (1970)
15. Korovin, K., Voronkov, A.: Orienting rewrite rules with the Knuth-Bendix order. *Information and Computation* 183, 165–186 (2003)
16. Kurihara, M., Kondo, H.: Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In: Orchard, B., Yang, C., Ali, M. (eds.) *IEA/AIE 2004*. LNCS (LNAI), vol. 3029, pp. 827–837. Springer, Heidelberg (2004)
17. Manquinho, V., Roussel, O.: Pseudo-boolean evaluation (2007)
<http://www.cril.univartois.fr/PB07/>
18. Marché, C.: Termination problem data base (TPDB), version 3.2, June 2006.
www.lri.fr/~marche/tpdb
19. Steinbach, J.: Extensions and comparison of simplification orders. In: Dershowitz, N. (ed.) *Rewriting Techniques and Applications*. LNCS, vol. 355, pp. 434–448. Springer, Heidelberg (1989)
20. Tseitin, G.: On the complexity of derivation in propositional calculus. In: *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pp. 115–125 (1968)
21. Zankl, H.: SAT techniques for lexicographic path orders. Seminar report (2006)
Available at <http://arxiv.org/abs/cs.SC/0605021>
22. Zankl, H., Hirokawa, N., Middeldorp, A.: Constraints for argument filterings. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) *SOFSEM 2007*. LNCS, vol. 4362, pp. 579–590. Springer, Heidelberg (2007)
23. Zankl, H., Middeldorp, A.: KBO as a satisfaction problem. In: *Proc. 8th International Workshop on Termination*, pp. 55–59 (2006)

Termination by Quasi-periodic Interpretations

Hans Zantema¹ and Johannes Waldmann²

¹ Department of Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

h.zantema@tue.nl

² Hochschule für Technik, Wirtschaft und Kultur (FH) Leipzig
Fb IMN, PF 30 11 66, D-04251 Leipzig, Germany
waldmann@imn.htwk-leipzig.de

Abstract. We present a new method for automatically proving termination of term rewriting and string rewriting. It is based on the well-known idea of interpretation of terms in natural numbers where every rewrite step causes a decrease. In the dependency pair setting only weak monotonicity is required for these interpretations. For these we use quasi-periodic functions. It turns out that then the decreasingness for rules only needs to be checked for finitely many values, which is easy to implement.

Using this technique we automatically prove termination of over ten string rewriting systems in TPDB for which termination was open until now.

1 Introduction

At the Workshop on Termination, Seattle, August 2006, termination of the string rewriting system (SRS) consisting of the two rules

$$aaa \rightarrow bab, \quad bbb \rightarrow aaa,$$

was presented as an open problem. Shortly after that Aleksey Nogin and Carl Witty came up with an ad hoc proof, based on observations on the length modulo 3 of maximal numbers of consecutive a 's in the string. Inspired by this proof based on this modulo 3 behavior, we found alternative termination proofs for the same SRS using dependency pairs and weakly monotone interpretations of the shape

$$f(x) = 3 * (x \text{ div } 3) = x - (x \text{ mod } 3).$$

In this way one challenge for the current paper was born: generalize this kind of functions in such a way that they are suitable for implementation, and that using these functions increases the power of termination provers.

During the last years proving termination of small SRSs received a lot of attention, see e.g. [\[14,5,13,10,6,8\]](#). Among the reasons for this we mention:

- It is an open problem whether termination of one rule string rewriting is decidable or not. Motivated by this open problem people are triggered to investigate termination behavior of single rules.

- There are several extremely small SRSs like $\{aabb \rightarrow bbbaaa\}$ and $\{aa \rightarrow bc, bb \rightarrow ac, cc \rightarrow ab\}$ for which proving termination turned out to be surprisingly hard. For these two examples termination proofs are known and can be found by tools, but for several others the termination problem is still open.

Every year there is a Termination Competition [9], where several tools are applied on the newest version of the Termination Problem Data Base (TPDB) [2]. In the string rewriting category of the 2006 version of TPDB there are 34 systems for which in the 2006 termination competition none of the tools could prove termination or non-termination, some of which are known to be non-terminating. By our implementation of the technique presented in this paper for 13 among these 34 systems termination proofs are generated fully automatically.

Our approach is an instance of the well-known theme of interpretations into a well-founded monotone algebra. It is well-known that in the setting of dependency pairs one only needs weak monotonicity in the algebra. In this paper we focus on the case where the monotone algebra simply consists of the natural numbers with the usual order. So the key point is to choose suitable weakly monotonic functions. In the usual approach these functions are always polynomials. Instead we choose functions that have a periodic difference with linear functions, more precisely, functions f satisfying

$$f(x + p) = f(x) + s * p$$

for all $x \in \mathbf{N}$, for some *period* $p \in \mathbf{N}$, $p > 1$ and some *slope* $s \in \mathbf{N}$. Such functions are called *quasi-periodic*. Note that the function f mentioned above given by $f(x) = 3 * (x \text{ div } 3)$ is an instance of a quasi-periodic function with period 3 and slope 1. Moreover, all linear polynomial interpretations are quasi-periodic.

The following observations make quasi-periodic functions suitable for use in automatic search for termination proofs:

- A quasi-periodic function is fully determined by its slope, its period, and finitely many values.
- Quasi-periodic functions are closed under composition, by which the interpretation of a term is quasi-periodic if the interpretations of its operations symbols are.
- Checking whether $\forall x : f(x) \geq g(x)$ for quasi-periodic functions f, g can be done by inspecting only finitely many values of x .

As a consequence, by fixing the period and bounding the slope the corresponding search space for function interpretations is finite. However, it may be huge. We have implemented the method by fixing the period and bounding the slope, transforming the search problem to a SAT problem, and using the state-of-the-art SAT solver minisat, version 2, [3]. Surprisingly, all newly solved SRSs from TPDB could be solved by fixing the slope to be 1. Therefore in the presentation we focus on this case, but we also implemented the approach for arbitrary bounded slopes.

This paper has been organized as follows. In Section 2 we recall the earlier theory on monotone algebras and dependency pairs as we need it, and we introduce quasi-periodic functions and investigate their basic properties. In Section 3 we work out these basic ingredients towards a proof scheme for string rewriting. In Section 4 we describe how this has been implemented by transforming the corresponding search problem to a SAT problem, and investigate some experimental results. In Section 5 we describe multi-dimensional quasi-periodic functions, and investigate how to use them for extending the approach to term rewriting. In Section 6 we describe a direct approach of using quasi-periodic functions without the use of the dependency pair transformation. Finally, in Section 7 we give some conclusions.

2 Basic Theory

For a term rewriting system (TRS) R we write \rightarrow_R for its rewrite relation and \xrightarrow{top}_R for its top rewrite relation, i.e., $t \xrightarrow{top}_R u$ if and only if there is a rewrite rule $\ell \rightarrow r \in R$ and a substitution σ such that $t = \ell\sigma$ and $u = r\sigma$.

A relation \rightarrow is called *well-founded*, *terminating* or *strongly normalizing*, notation $\text{SN}(\rightarrow)$, if no infinite sequence t_1, t_2, t_3, \dots exists such that $t_i \rightarrow t_{i+1}$ for all $i = 1, 2, 3, \dots$. A TRS R is called *terminating* if $\text{SN}(\rightarrow_R)$, shortly written as $\text{SN}(R)$.

A binary relation \rightarrow_1 is called *terminating relative to a binary relation* \rightarrow_2 , written as $\text{SN}(\rightarrow_1 / \rightarrow_2)$, if no infinite sequence t_1, t_2, t_3, \dots exists such that

- $t_i \rightarrow_1 t_{i+1}$ for infinitely many values of i , and
- $t_i \rightarrow_2 t_{i+1}$ for all other values of i .

We use the notation $\rightarrow_1 / \rightarrow_2$ to denote $\rightarrow_2^* \cdot \rightarrow_1 \cdot \rightarrow_2^*$; it is easy to see that $\text{SN}(\rightarrow_1 / \rightarrow_2)$ coincides with well-foundedness of $\rightarrow_1 / \rightarrow_2$. We write $\text{SN}(R/S)$ as a shorthand for $\text{SN}(\rightarrow_R / \rightarrow_S)$, and we write $\text{SN}(R_{top}/S)$ as a shorthand for $\text{SN}(\xrightarrow{top}_R / \rightarrow_S)$.

For a TRS R over a signature Σ a symbol $f \in \Sigma$ is called a *defined symbol* if f is the root symbol of a left-hand side of a rule of R . For every defined symbol $f \in \Sigma$ a new marked symbol $f_\#$ is added having the same arity as f . If $f(s_1, \dots, s_n) \rightarrow C[g(t_1, \dots, t_m)]$ is a rule in R and g is a defined symbol of R , then the rewrite rule $f_\#(s_1, \dots, s_n) \rightarrow g_\#(t_1, \dots, t_m)$ is called a *dependency pair* of R . The TRS consisting of all dependency pairs of R is denoted by $\text{DP}(R)$.

The main theorem about dependency pairs is the following, due to Arts and Giesl, [1].

Theorem 1. *Let R be a TRS. Then $\text{SN}(R)$ if and only if $\text{SN}(\text{DP}(R)_{top}/R)$.*

In order to prove termination of a TRS R we will prove $\text{SN}(\text{DP}(R)_{top}/R)$. In order to do so we describe a technique for proving $\text{SN}(R_{top}/S)$ for arbitrary TRSs R, S , based on weakly monotone algebras.

Definition 1. A Σ -algebra $(A, [\cdot])$ is defined to consist of a non-empty set A , and for every $f \in \Sigma$ a function $[f] : A^n \rightarrow A$, where n is the arity of f . This function $[f]$ is called the interpretation of f .

A operation $[f] : A^n \rightarrow A$ is monotone with respect to a binary relation \rightarrow on A if for all $a_i, b_i \in A$ for $i = 1, \dots, n$ with $a_i \rightarrow b_i$ for some i and $a_j = b_j$ for all $j \neq i$ we have

$$[f](a_1, \dots, a_n) \rightarrow [f](b_1, \dots, b_n).$$

A weakly monotone Σ -algebra $(A, [\cdot], >, \succsim)$ is a Σ -algebra $(A, [\cdot])$ equipped with two relations $>, \succsim$ on A such that

- $>$ is well-founded;
- $> \cdot \succsim \subseteq >$;
- for every $f \in \Sigma$ the operation $[f]$ is monotone with respect to \succsim .

The combination $>, \succsim$ is closely related to the notion of *reduction pair* as presented e.g. in [7]. A crucial difference is that the relations in a reduction pair are relations on terms that are closed under substitutions, while in our setting they are relations on the set A .

Writing \mathcal{X} for the set of variables, for a Σ -algebra $(A, [\cdot])$ and a map $\alpha : \mathcal{X} \rightarrow A$ the term evaluation $[\cdot, \alpha] : \mathcal{T}(\Sigma, \mathcal{X}) \rightarrow A$ is defined inductively by

$$[x, \alpha] = \alpha(x), \quad [f(t_1, \dots, t_n), \alpha] = [f]([t_1, \alpha], \dots, [t_n, \alpha])$$

for $f \in \Sigma$ and $x \in \mathcal{X}$.

As the main property of weakly monotone algebras we recall the following theorem from [4].

Theorem 2. Let R, S be TRSs over a signature Σ . Let $(A, [\cdot], >, \succsim)$ be a weakly monotone Σ -algebra such that $[l, \alpha] \succsim [r, \alpha]$ for every rule $l \rightarrow r$ in $R \cup S$ and every $\alpha : \mathcal{X} \rightarrow A$. Let R' consist of all rules $l \rightarrow r$ from R satisfying $[l, \alpha] > [r, \alpha]$ for every $\alpha : \mathcal{X} \rightarrow A$.

Then $\text{SN}((R \setminus R')_{\text{top}}/S)$ if and only if $\text{SN}(R_{\text{top}}/S)$.

The approach for proving $\text{SN}(R)$ now is trying to prove $\text{SN}(\text{DP}(R)_{\text{top}}/R)$ by finding a suitable weakly monotone algebra such that according to Theorem [2] rules from $\text{DP}(R)$ may be removed. This is repeated until all rules of $\text{DP}(R)$ have been removed.

We will focus on $A = \mathbf{N}$, where $>$ is the usual ordering on \mathbf{N} and \succsim coincides with \geq . Now indeed $>$ is well-founded and $> \cdot \succsim \subseteq >$ holds, so the only requirement for being a weakly monotone algebra is that for every $f \in \Sigma$ the operation $[f]$ is monotone with respect to \geq . Monotonicity with respect to \geq is called *weak monotonicity*.

Definition 2. A function $f : \mathbf{N} \rightarrow \mathbf{N}$ is called quasi-periodic with period $p \in \mathbf{N}$, $p > 0$, and slope $s \in \mathbf{N}$ if $f(x + p) = f(x) + s * p$ for all $x \in \mathbf{N}$.

For $[f]$ we will use quasi-periodic functions in case f is unary; for f of higher arity we will extend this notion in Section [5].

Theorem 3. *Let $f : \mathbf{N} \rightarrow \mathbf{N}$ be a quasi-periodic function with period $p \in \mathbf{N}$, $p > 0$, and slope $s \in \mathbf{N}$. Then*

1. $f(x + n * p) = f(x) + n * s * p$ for all $x, n \in \mathbf{N}$.
2. f is weakly monotone if and only if $f(x + 1) \geq f(x)$ for $x = 0, \dots, p - 1$.
3. Let $g : \mathbf{N} \rightarrow \mathbf{N}$ be a quasi-periodic function with period p and slope t . Then $f \circ g$ is quasi-periodic with period p and slope $s * t$.
4. Let $g : \mathbf{N} \rightarrow \mathbf{N}$ be a quasi-periodic function with period p and slope t . Then $f(x) \geq g(x)$ for all $x \in \mathbf{N}$ if and only if $s \geq t$ and $f(x) \geq g(x)$ for $x = 0, \dots, p - 1$.

Proof. 1. Induction on n .

2. By definition f is weakly monotone if and only if $f(x + i) \geq f(x)$ for all $x, i \in \mathbf{N}$; by induction on i this is equivalent to $f(x + 1) \geq f(x)$ for all x . For every $x \in \mathbf{N}$ we can write $x = x' + n * p$ for some $n \in \mathbf{N}$ with $x' < p$. Using part (1) then yields $f(x + 1) - f(x) = f(x' + 1) - f(x')$ from which the claim follows.
3. Using $t \in \mathbf{N}$ and part (1) we obtain

$$\begin{aligned} (f \circ g)(x + p) &= f(g(x + p)) \\ &= f(g(x) + t * p) \\ &= f(g(x)) + s * t * p \\ &= (f \circ g)(x) + s * t * p \end{aligned}$$

4. For the ‘if’-part we write $x = x' + n * p$ with $x' < p$ and use part (1):

$$\begin{aligned} f(x) - g(x) &= f(x') + n * s * p - (g(x') + n * t * p) \\ &= f(x') - g(x') + n * (s - t) * p \\ &\geq 0 \end{aligned}$$

For the ‘only if’-part we assume $f(x) \geq g(x)$ for all $x \in \mathbf{N}$ and need to show that $s \geq t$. Assume not, then $t \geq s + 1$ and $f(n * p) - g(n * p) = f(0) - g(0) + n * (s - t) * p \leq f(0) - g(0) - n * p < 0$ for n large enough, contradiction. □

A quasi-periodic function with period p is also quasi-periodic with period $n * p$ for $n \in \mathbf{N}$, $n > 0$. Using part (3) of Theorem 3 one sees that composition of quasi-periodic functions with distinct periods is again quasi-periodic, with the period being the least common multiple of both periods. However, we will not use this: in every setting we fix all periods to be the same.

One may also allow non-integer slopes. For instance, f defined by $f(x) = 2 * (x \text{ div } 3)$ can be seen as quasi-periodic with period 3 and slope $\frac{2}{3}$. However, quasi-periodic functions with a fixed period and non-integer slope are not closed under composition. For instance, for this function f the function $f \circ f$ does not have period 3 any more. It has period 9 and slope $\frac{4}{9}$. Since for our purpose closedness under composition is essential we restrict to integer non-negative slopes. We allow the slope to be 0; this corresponds to periodic functions. A quasi-periodic function with slope 0 is weakly monotone if and only if it is constant.

3 String Rewriting

String rewriting can be seen as the special case of term rewriting in which all symbols are unary. In this case we need only one variable x , and we identify the string $a_1 \cdots a_n$ with the term $a_1(\cdots(a_n(x))\cdots)$. We write α_i for the map mapping this variable x to the number i , for the rest we may forget about this variable.

We propose the following approach for proving termination of an SRS S .

Compute $\text{DP}(S)$ and try to prove $\text{SN}(\text{DP}(S)_{\text{top}}/S)$; if we succeed then indeed by Theorem 1 we may conclude $\text{SN}(S)$. Trying to prove $\text{SN}(R_{\text{top}}/S)$ for an SRS R is done as follows. First try to remove rules from R by Theorem 2 using simple polynomials (or any other fast technique). Next apply the following proof scheme:

Proof scheme.

- Fix a period $p \geq 1$.
- For every symbol a choose a slope $\text{slope}(a)$ and p natural numbers $[a](x)$ for $x = 0, \dots, p-1$, fully defining $[a] : \mathbf{N} \rightarrow \mathbf{N}$ by $[a](x+p) = [a](x) + p * \text{slope}(a)$, meeting the following requirements:
 - $[a](x+1) \geq [a](x)$ for every symbol a and every $x = 0, \dots, p-1$ (where $[a](p) = [a](0) + p * \text{slope}(a)$),
 - $[\ell, \alpha_i] \geq [r, \alpha_i]$ for all rules $\ell \rightarrow r$ in $R \cup S$ and all $i = 0, \dots, p-1$,
 - for at least one rule $\ell \rightarrow r$ in R it holds that $[\ell, \alpha_i] > [r, \alpha_i]$ for all $i = 0, \dots, p-1$; write R' for the rules from R for which this holds.
- If $R' = R$ we are done, otherwise $\text{SN}((R \setminus R')_{\text{top}}/S)$ has to be proved, either by repeating this proof scheme or by any other technique.

Correctness of this approach follows by combining Theorem 2 and Theorem 3. For $p = 1$ the scheme coincides with linear polynomials.

Example 1. We consider the example mentioned in the introduction: the two rules $aaa \rightarrow bab$, $bbb \rightarrow aaa$. Applying Theorem 1 we will prove termination of this SRS S by proving $\text{SN}(\text{DP}(S)_{\text{top}}/S)$. By considering lengths of strings, more precisely, by applying Theorem 2 where the algebra consists of the natural numbers and every symbol is interpreted by the successor function, it remains to prove $\text{SN}(R_{\text{top}}/S)$ for R consisting of the two rules $Aaa \rightarrow Bab$, $Bbb \rightarrow Aaa$, where we simply write A, B instead of the marked symbols $a_{\#}$ and $b_{\#}$. For this we apply the above approach: we fix $p = 3$, all slopes are one, and choose

$$\begin{aligned} [a](0) &= 2, & [b](0) &= 3, & [A](0) &= 2, & [B](0) &= 3, \\ [a](1) &= 3, & [b](1) &= 3, & [A](1) &= 4, & [B](1) &= 4, \\ [a](2) &= 4, & [b](2) &= 3, & [A](2) &= 5, & [B](2) &= 4. \end{aligned}$$

All properties are checked for R' consisting of the rule $Bbb \rightarrow Aaa$. So this rule may be removed from R . The remaining property $\text{SN}(\{Aaa \rightarrow Bab\}_{\text{top}}/S)$ is easily proved by counting the number of A symbols.

We conclude this section by some basic transformations on SRSs preserving termination.

For a string s write s^{rev} for its reverse. For an SRS R write

$$R^{\text{rev}} = \{ \ell^{\text{rev}} \rightarrow r^{\text{rev}} \mid \ell \rightarrow r \in R \}.$$

and

$$R^{-1} = \{ r \rightarrow \ell \mid \ell \rightarrow r \in R \}.$$

From [13] we recall the following simple lemma.

Lemma 1. *Let R be an SRS.*

1. $\text{SN}(R)$ if and only if $\text{SN}(R^{\text{rev}})$.
2. Assume that R is finite and that for every rule $\ell \rightarrow r$ of R the lengths of ℓ and r are equal. Then $\text{SN}(R)$ if and only if $\text{SN}(R^{-1})$.

Slightly more involved, and increasing the SRS size, is the following transformation. For a set Σ of symbols we define $\text{lab} : \Sigma^* \times \Sigma \rightarrow (\Sigma \times \Sigma)^*$ as follows:

$$\text{lab}(\epsilon, a) = \epsilon, \quad \text{lab}(sa, b) = \text{lab}(s, a)(a, b),$$

for all $a, b \in \Sigma, s \in \Sigma^*$. Here ϵ denotes the empty string. For an SRS R over Σ we define

$$\text{lab}(R) = \{ \text{lab}(\ell, a) \rightarrow \text{lab}(r, a) \mid \ell \rightarrow r \in R \wedge a \in \Sigma \}.$$

For a non-empty string s write s_1 for its first element. In order to force that for every rule $\ell \rightarrow r$ we have $\ell_1 = r_1$, for arbitrary SRS R over Σ we define the SRS $F(R)$ to be

$$\{ \ell \rightarrow r \in R \mid r \neq \epsilon \wedge \ell_1 = r_1 \} \cup \{ a\ell \rightarrow ar \mid a \in \Sigma \wedge \ell \rightarrow r \in R \wedge (r = \epsilon \vee \ell_1 \neq r_1) \}.$$

Lemma 2. *Let R be an SRS. Then $\text{SN}(R)$ if and only if $\text{SN}(\text{lab}(F(R)))$.*

Proof. Equivalence of $\text{SN}(R)$ and $\text{SN}(F(R))$ is straightforward. Equivalence of $\text{SN}(F(R))$ and $\text{SN}(\text{lab}(F(R)))$ is a direct application of semantic labelling [11] in which the model is Σ and every symbol is interpreted by its own value. For the model requirement it is essential that for every rule $\ell \rightarrow r$ in $F(R)$ we have $\ell_1 = r_1$. \square

The idea of these transformations is that if proving termination of the original system fails, then proving termination of the transformed system is tried. This idea is not new. For rev it is extensively used in several tools; for $\text{lab}(F(\cdot))$ it was extensively and very successfully used in the 2006 competition by Jambox, see [9]. In Example 3 we will see how Lemma 2 can be applied fruitfully.

4 Implementation and Results

We experimented with various ways for automatically finding termination proofs based on the above given proof scheme. The first one simply chose several times randomly among a class of quasi-periodic functions until all requirements were fulfilled. The class of quasi-periodic functions consisted of functions of the shape $\lambda x \cdot x + n$ and $\lambda x \cdot p * (x \text{ div } p) + n$ for constants n . In this way the first automatically found termination proof for the example in the introduction was given, only a few days after Nogin and Witty found their manual proof.

However, this random search has a few drawbacks:

- It depends on the special choice of the shape of the quasi-periodic functions chosen, being much more restricted than arbitrary quasi-periodic functions with fixed period and slope.
- If no proof is found you do not know whether no proof of the desired shape exists or you did not yet try long enough.

A remedy against both these drawbacks is the following. For all numbers chosen in the proof scheme choose numbers in binary notation, and introduce boolean variables for each of the bits. Express all requirements in the proof scheme as propositional requirements on these boolean variables. Then a choice of the numbers satisfying all requirements is possible if and only if the formula is satisfiable. So the approach is to apply a state-of-the-art SAT solver to the resulting formula (just like for several other methods for proving termination), and in case the formula is satisfiable transform the bits of the numbers in the corresponding satisfying assignment back to the numbers they represent.

Still the encoding of the requirements can be done in several ways. One fruitful way is the following. It easily extends to arbitrary slopes; for keeping the presentation simple we restrict here to the case where all slopes are one. Assume we want to prove $\text{SN}(R_{\text{top}}/S)$.

We fix three numbers: p is the period, n is the number of bits per number, by which all numbers are non-negative and $< 2^n$, and m is the maximal number allowed as an intermediate result, satisfying $p < m < 2^n - p$. For all symbols a we choose $m * n$ boolean variables for the m n -bit numbers $[a](0), \dots, [a](m-1)$. For these we generate the requirements $[a](i) \leq [a](i+1)$ for $i = 0, \dots, p-1$ and $[a](i+p) = [a](i) + p$ for $i = 0, \dots, m-p-1$. Having all these numbers $[a](0), \dots, [a](m-1)$ available in separate n -bit notation makes it straightforward to express the boolean formula $[a](i) = j$ for given numbers $i, j, i < m$.

In order to check $[\ell, \alpha_i] \geq [r, \alpha_i]$, for every rule $\ell \rightarrow r$ in $R \cup S$ and every $i = 0, \dots, p-1$ the following is done. Write $\ell = a_1 a_2 \dots a_k$, then k fresh binary numbers $\ell_{i,1}, \dots, \ell_{i,k}$ are introduced, for which the following requirements are created:

$$\ell_{i,k} = [a_k](i),$$

$$(j = \ell_{i,q}) \rightarrow (\ell_{i,q-1} = [a_q](j)), \text{ for all } q = 1, \dots, k, j = 0, \dots, m-1,$$

$$m > \ell_{i,q}, \text{ for all } q = 2, \dots, k.$$

This is done similarly for r . Next the requirement $\ell_{i,1} \geq r_{i,1}$ is generated. Finally the requirement is added that for at least one rule in R we have $\ell_{i,1} > r_{i,1}$.

A remarkable property of this approach is that the only arithmetic that occurs in these formulas is the unary function $\lambda x \cdot x + p$ and checking for $>$ and \geq . As a consequence, the resulting formulas are relatively small. For instance, our solution given in Example [1](#) was found by applying SAT on a formula consisting of 6065 clauses on 872 variables, in a fraction of a second, after choosing the parameters $p = 3$, $n = 4$, $m = 7$.

In order to find termination proofs fully automatically, the approach should not depend on parameters that have to be set manually. Therefore we need heuristics for setting these parameters and for how to combine this with other techniques.

We chose always to combine this with basic polynomials, more precisely, apply Theorem [2](#) with natural numbers and linear polynomials with coefficients in n bits, and try both these basic polynomials and the quasi-periodic interpretations with the fixed parameters as long as possible. Moreover, we apply Lemma [3](#) after an attempt fails for an SRS S , then a next attempt is done for S^{rev} . If even that fails, then for length-preserving systems the same is done for S^{-1} .

Of course our approach is easily combined with other termination techniques, but in order to investigate the merits of the technique itself we concentrated on running the experiments in this most basic setting.

After fixing this basic setting, a choice should be made for the parameters p, n, m . We experimented with these parameters on several small SRSs for which termination was unknown until now.

It appears that increasing these parameters never decreases the power. For the parameters n, m this is obvious; for the period p this is only indicated by experiments: we failed to prove this. For one system, Gebhardt18 in TPDB, consisting of the two rules $0000 \rightarrow 1011$, $1001 \rightarrow 0000$, it turned out that no proof was found with period less than 5, but a proof was found for period 5. We did not find examples that could be solved with period higher than 5 but not by period 5. So a suitable choice for the period is 5.

It turned out that all proofs we found could be found using numbers of four bits. A corresponding choice of the parameters is $n = 4$ and $m = 10$. Fixing these parameters $p = 5$, $n = 4$ and $m = 10$ we found termination proofs of the following 11 SRSs in TPDB 2006, of which in the 2006 competition none of the participating tools found a termination proof:

- in the directory Endrullis: systems 01, 02, 05 and 06,
- in the directory Gebhardt: systems 01, 04, 07, 11, 17 and 18,
- in the directory Waldmann: system jw1 (this is the system from Example [1](#)).

All of these systems are very small: each one consists of only two rules over two symbols. In all of these cases finding the satisfying assignment of the SAT formula representing the quasi-periodic interpretation was done by minisat within a fraction of a second.

Example 2. As an example we give the proof of the system Gebhardt18 as shown above, found in this way by our implementation with the parameters $p = 5$, $n = 4$

and $m = 10$. After removing the length-decreasing dependency pairs it remains to prove $\text{SN}(R_{\text{top}}/S)$ for R consisting of the rules $0_{\#}000 \rightarrow 1_{\#}011$, $1_{\#}001 \rightarrow 0_{\#}000$ and S consisting of the original rules $0000 \rightarrow 1011$, $1001 \rightarrow 0000$. For that the following quasi-periodic interpretation with period 5 is found:

$$\begin{aligned} 0 &= 1, & [0](1) &= 2, & [0](2) &= 3, & [0](3) &= 4, & [0](4) &= 5, \\ [1](0) &= 1, & 1 &= 1, & [1](2) &= 1, & [1](3) &= 6, & [1](4) &= 6, \\ [0_{\#}](0) &= 2, & [0_{\#}](1) &= 7, & [0_{\#}](2) &= 7, & [0_{\#}](3) &= 7, & [0_{\#}](4) &= 7, \\ [1_{\#}](0) &= 4, & [1_{\#}](1) &= 4, & [1_{\#}](2) &= 4, & [1_{\#}](3) &= 8, & [1_{\#}](4) &= 8. \end{aligned}$$

As a consequence the rule $1_{\#}001 \rightarrow 0_{\#}000$ may be removed from R , after which the rest is trivial by counting the number of $0_{\#}$ symbols.

This implementation restricts to the case where all slopes are equal to one. The implementation easily extends to setting with arbitrary slopes. We also experimented with other encodings of the problem, including general slopes. Surprisingly, using general slopes we only found termination proofs for systems for which also proofs with slope one were found. Hence generalizing the slope does not seem to increase the power of the method, and therefore we do not describe the implementation for general slopes in more detail.

Example 3. As an example of a combination with other techniques we consider the special case of semantic labelling ([11]) as described in Lemma 2. Applying $\text{lab}(F(\cdot))$ to the SRS Gebhardt09, consisting of the two rules $0000 \rightarrow 0111$, $1001 \rightarrow 0000$, yields the transformed system of six rules over the four symbols 00, 01, 10 and 11:

$$\begin{array}{ll} 00\ 00\ 00\ 00 \rightarrow 01\ 11\ 11\ 10, & 00\ 00\ 00\ 01 \rightarrow 01\ 11\ 11\ 11, \\ 01\ 10\ 00\ 01\ 10 \rightarrow 00\ 00\ 00\ 00\ 00, & 01\ 10\ 00\ 01\ 11 \rightarrow 00\ 00\ 00\ 00\ 01, \\ 11\ 10\ 00\ 01\ 10 \rightarrow 10\ 00\ 00\ 00\ 00, & 11\ 10\ 00\ 01\ 11 \rightarrow 10\ 00\ 00\ 00\ 01. \end{array}$$

It turns out that by the technique described in this paper termination of this transformed SRS (and hence of the original SRS by Lemma 2) is easily proved, again automatically generating a termination proof of an SRS for which termination was open until now.

A similar termination proof was found for the SRS Waldmann-sym-5, consisting of the two rules $aaa \rightarrow bbb$, $bbbb \rightarrow abba$. For this SRS termination was open until now too, by which the total score of new automatic proofs for unsolved SRSs from TPDB becomes 13.

5 Term Rewriting

In order to apply our method to term rewriting, we call a function $f : \mathbf{N}^k \rightarrow \mathbf{N}$ *quasi-periodic* with period $p \in \mathbf{N}, p > 0$ and slope $s = (s_1, \dots, s_k) \in \mathbf{N}^k$ if for each $x \in \mathbf{N}^k$ and each $1 \leq i \leq k$

$$f(x_1, \dots, x_i + p, \dots, x_k) = f(x_1, \dots, x_i, \dots, x_k) + s_i * p.$$

That is, f is quasi-periodic with slope s_i in its i -th argument.

The analogue of Theorem 3 still holds, with similar proof:

Theorem 4. *Let $f : \mathbb{N}^k \rightarrow \mathbb{N}$ be a quasi-periodic function with period $p \in \mathbb{N}, p > 0$ and slope $s = (s_1, \dots, s_k) \in \mathbb{N}^k$. Then*

1. *f is weakly monotonic if for all $(x_1, \dots, x_k) \in \{0, 1, \dots, p - 1\}^k$ and for all $1 \leq i \leq k$ we have $f(x_1, \dots, x_i + 1, \dots, x_k) \geq f(x_1, \dots, x_i, \dots, x_k)$.*
2. *Let $g_1, \dots, g_k : \mathbb{N}^l \rightarrow \mathbb{N}$ be quasi-periodic functions with period p and slopes t_1, \dots, t_k , respectively. Then the combined function $h : \mathbb{N}^l \rightarrow \mathbb{N}$ defined by $h(x) = f(g_1(x), \dots, g_k(x))$ is quasi-periodic with period p and its slope in the i -th position is $s_1 t_{1,i} + \dots + s_k t_{k,i}$.*
3. *Let $g : \mathbb{N}^k \rightarrow \mathbb{N}$ be a quasi-periodic function with period p and slope $t = (t_1, \dots, t_k)$. Then $f(x) \geq g(x)$ for all $x \in \mathbb{N}^k$ if and only if $s_1 \geq t_1, \dots, s_k \geq t_k$ and $f(x) \geq g(x)$ for all $x \in \{0, 1, \dots, p - 1\}^k$.*

Corollary 1. *If an interpretation $[\cdot]$ is given that assigns to each k -ary function symbol f from the signature Σ a k -ary quasi-periodic function $[f]$ with period p , then the interpretation $[t]$ of any term t containing n variables is an n -ary quasi-periodic function with period p .*

Proof. If t is a variable v_i , then $[t]$ is the projection $(v_1, \dots, v_n) \mapsto v_i$ which is quasi-periodic with slope $(0, \dots, 0, 1, 0 \dots 0)$ and any period $p > 0$. If $t = f(t_1, \dots, t_k)$, then the interpretation can be composed (by Theorem 4) from the interpretations $[t_1], \dots, [t_k]$. This includes the case $k = 0$. □

To find a quasi-periodic interpretation that proves (relative) termination of a given term rewriting system, we extend the SAT solver approach mentioned in Section 4. In the following, the word *variable* means "a sequence of Boolean variables that represents a natural number".

We represent a k -ary quasi-periodic function f of period p by k variables that represent the slopes (s_1, \dots, s_k) and p^k variables that represent $f(x)$ for $x \in \{0, 1, \dots, p - 1\}^k$. We ensure that f is weakly monotonic by $p^k \times k$ constraints according to Theorem 4, part 1.

With the notation of Part 2 of that theorem, the combined function h has each component of its slope vector constrained by one equation. To constrain the (initial) values of h , for each argument tuple $x \in \{0, \dots, p - 1\}^l$, and each $i \in \{1, \dots, k\}$, we use a pair of variables (q_i, r_i) (quotient and remainder) that fulfill $g_i(x) = p * q_i + r_i \wedge r_i < p$. Then the values of h must obey the constraint $h(x) = s_1 q_1 + \dots + s_k q_k + f(r_1, \dots, r_k)$.

Our implementation is restricted to the case where the period p is a power of two because then the construction of q_i, r_i is much simpler, since all variables use a base two representation.

We illustrate the method by the following examples.

Example 4. From the rewriting system

$$R = \{ c(f(y, b(f(0, x)))) \rightarrow f(c(c(a(f(y, x))))), y), \\ f(0, b(f(y, x))) \rightarrow b(b(y)), f(y, a(f(0, x))) \rightarrow y, a(c(y)) \rightarrow y \}$$

the essential part of the DP transformed problem is $\text{SN}(S_{\text{top}}/R)$ with $S = \{C(f(y, b(f(0, x)))) \rightarrow C(c(a(f(y, x))))\}, C(f(y, b(f(0, x)))) \rightarrow C(a(f(y, x)))\}$.

Now we use a quasi-periodic interpretation of period 2.

$$\begin{aligned} [0] &= 0, & [a](x) &= \lfloor x \rfloor_2, & [b](x) &= 0 \\ [c](x) &= \lfloor x + 1 \rfloor_2, & [f](x, y) &= \lfloor x \rfloor_2 + 1, & [C](x) &= x. \end{aligned}$$

Here, $\lfloor x \rfloor_m$ denotes $m * (x \text{ div } m) = x - (x \bmod m)$ (round down to the next multiple of m). This interpretation is weakly compatible with R (meaning that $[\ell, \alpha] \geq [r, \alpha]$ for all α and all rules $\ell \rightarrow r$), for instance, the interpretation of $a(c(y))$ is $\lfloor y + 1 \rfloor_2$, which is equal to y for even y and greater than y for odd y ; and strictly compatible with S (meaning that $[\ell, \alpha] > [r, \alpha]$ for all α and all rules $\ell \rightarrow r$), for instance, the interpretation of $C(f(y, b(f(0, x))))$ is $\lfloor y \rfloor_2 + 1$ and the interpretation of $C(a(f(y, x)))$ is $\lfloor y \rfloor_2$.

Example 5. For the rewriting system

$$\begin{aligned} R = \{ & a(c(a(c(c(f(y, 0)))))) \rightarrow y, & f(y, f(0, c(x))) &\rightarrow a(c(y)), \\ & f(y, x) \rightarrow c(c(y)), & c(a(f(y, f(0, x)))) &\rightarrow a(f(f(f(y, y), x), 0)) \} \end{aligned}$$

the essential part of $\text{DP}(R)$ is

$$\begin{aligned} S = \{ & F(y, f(0, c(x))) \rightarrow C(y), & F(y, x) &\rightarrow \{C(c(y)), C(y)\}, \\ & C(a(f(y, f(0, x)))) &\rightarrow \{F(f(f(y, y), x), 0), & F(f(y, y), x), F(y, y)\} \}, \end{aligned}$$

where we have grouped together rules with identical left-hand sides. Then the following quasi-periodic interpretation

$$\begin{aligned} [0] &= 0, [a](x) = x + 1, [c](x) = \lfloor x \rfloor_2, [C](x) = \lfloor x \rfloor_2 + 2, \\ [f](x, y) &= \lfloor x \rfloor_2 + 1, [F](x, y) = \lfloor x \rfloor_2 + 2 \end{aligned}$$

is weakly compatible with $R \cup S$ and strictly compatible with those three rules from S that have C as top symbol. So they can be removed, and the remaining problem is easily solved.

None of the above examples can be solved by Jambox or Aprove (2006 version). In our implementation the actual construction and solution (by minisat) of the constraint system only takes a few seconds.

Both systems in the above examples are non-linear. A quasi-periodic interpretation that is compatible with a non-linear system cannot have all slopes equal to one. The examples also indicate that it is quite powerful to extend the range of slopes to just $\{0, 1\}$.

The dependency pairs transformation creates rewriting systems that consist of groups of rules with identical left-hand sides. So the computation of the interpretation of left-hand sides can be shared. In fact, our implementation also shares interpretations of identical subterms (in all rules). This leads to a substantial reduction of the size of the constraint system and the run time of the solver.

6 Direct Interpretations

Until now the dependency pairs transformation is an essential ingredient of our approach. This has a few drawbacks: in this way the approach does not apply for relative termination, and neither serves for investigation of derivational complexity or more restricted variants of termination like total termination. Therefore in this section we investigate how quasi-periodic interpretations can be used directly without dependency pairs transformation. For that we need strict monotonicity rather than weak monotonicity, and the classical monotone algebra approach [12]. A quasi-periodic function $f : \mathbf{N} \rightarrow \mathbf{N}$ is called *strictly monotonic* if $x < y$ implies $f(x) < f(y)$.

Note that a strictly monotonic quasi-periodic function has slope ≥ 1 , and the only such functions of slope = 1 are the functions $x \mapsto x + c$. The function $x \mapsto 1 + x + \lfloor x \rfloor_2$ is strictly monotonic and has slope 2 and period 2. Here again $\lfloor x \rfloor_m$ denotes $m * (x \text{ div } m)$. We have the following properties:

- Proposition 1.** – *A quasi-periodic function f of period p is strictly monotonic if and only if we have $f(x) < f(x + 1)$ for every $x \in \{0, 1, \dots, p - 1\}$.*
- *If both f and g are strictly monotonic and quasi-periodic of period p , then the composition $x \mapsto f(g(x))$ is strictly monotonic.*

This means that we can handle strictly monotonic quasi-periodic functions effectively.

The classical monotone algebra approach restricting to natural numbers states that for proving $\text{SN}(R/S)$ it suffices to give an interpretation $[\]$ that assigns to each letter of the signature a strictly monotonic function, such that $[\ell](x) > [r](x)$ for each rule $\ell \rightarrow r \in R$ and $[\ell](x) \geq [r](x)$ for each rule $\ell \rightarrow r \in S$, for all $x \in \mathbf{N}$.

This scheme can be implemented in the same way as for weakly monotonic functions. This approach may give very simple termination proofs.

Example 6. Consider the strictly monotonic interpretation of period 2

$$[a](x) = 1 + x + \lfloor x \rfloor_2, \quad [b](x) = 2x.$$

It is strictly compatible with $R = \{a^3 \rightarrow bba, aba \rightarrow bba\}$ and weakly compatible with $S = \{bab \rightarrow aab\}$. This can be checked by a finite case analysis:

	$x \lfloor a^3 \rfloor(x)$	$\lfloor aba \rfloor(x)$	$\lfloor bba \rfloor(x)$	$\lfloor bab \rfloor(x)$	$\lfloor aab \rfloor(x)$
0	5	5	4	2	2
1	10	9	8	10	10

This proves $\text{SN}(R/S)$. We also have $\text{SN}(S)$, by counting letters a , therefore $\text{SN}(R \cup S)$, which is the SRS Bouchare-06 in TPDB.

This problem has been solved in the 2006 competition by some of the tools, but all generated proofs are much more complicated than this simple proof.

Since any quasi-periodic function is bounded from above by a linear function, the interpretation of a word then is bounded by an iterated linear function,

giving an exponential function. Thus if a strictly monotonic quasi-periodic interpretation is compatible with a rewriting system (removes all rules at the same time), then the derivational complexity of that system is at most exponential. We further note that since our interpretation domain is $(\mathbf{N}, >)$, which is totally ordered, we may conclude total termination. This remains true if a proof is given by repeated application of this direct method, since the lexicographic product of a number of copies of \mathbf{N} is still totally ordered. Hence Example 6 shows total termination of the SRS Bouchare-06 in TPDB.

7 Conclusions

We introduced a new technique for proving termination of both term rewriting and string rewriting. For both categories we succeeded in giving examples where our technique applies and earlier techniques fail.

In particular, by our technique we proved termination of several SRSs in TPDB for which termination was open until now. All of them consist of only two rules over two symbols. The reason for this is simple: most of the SRSs in TPDB for which termination is open came out of extensive search among randomly generated SRSs of this shape, filtered by failure of earlier tools.

This does not mean that restricted to string rewriting our approach is only successful for SRSs of this very special shape. For instance, our approach easily finds a termination proof for the SRS consisting of the following ten rules

$$ab \rightarrow cd, \quad cc \rightarrow bd, \quad b \rightarrow cf, \quad dd \rightarrow g, \quad cd \rightarrow h,$$

$$f \rightarrow g, \quad f \rightarrow dd, \quad gh \rightarrow ac, \quad hg \rightarrow f, \quad a \rightarrow dc,$$

where all other approaches until now fail.

In order to investigate the primary merits of our approach we focused our experiments on the most basic setting: only combine it with the most basic kind of polynomials and, for string rewriting, reversal of rules. For this basic setting we showed that our approach is successful: we found termination proofs of 11 systems in TPDB for which termination was open until now. This is definitely not the end point: this shows that it makes sense to plug in our technique in other tools for proving termination, in order to combine it with the wide range of termination techniques that have been implemented until now. Of course we appreciate that this will not be done before publication of this paper. As a first step in this direction we showed that by combining our approach with a very basic instance of semantic labelling for string rewriting, termination could be proved of two more SRSs in TPDB for which termination was open until now.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. *Theoretical Computer Science* 236, 133–178 (2000)
2. Termination Problems Data Base <http://www.lri.fr/~marche/tpdb/>

3. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
Tool: <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/>
4. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 574–588. Springer, Heidelberg (2006)
5. Geser, A.: Termination of string rewriting rules that have one pair of overlaps. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 410–423. Springer, Heidelberg (2003)
6. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: Finding finite automata that certify termination of string rewriting. *International Journal of Foundations of Computer Science* 16(3), 471–486 (2005)
7. Giesl, J., Arts, T., Ohlebusch, E.: Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation* 34(1), 21–58 (2002)
8. Hofbauer, D., Waldmann, J.: Proving termination with matrix interpretations. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 328–342. Springer, Heidelberg (2006)
9. Marché, C., Zantema, H.: The termination competition. In: Baader, F. (ed.) Proceedings of the 18th Conference on Rewriting Techniques and Applications (RTA), Springer, Heidelberg (2007)
<http://www.lri.fr/~marche/termination-competition/>
10. Moczydłowski, W., Geser, A.: Termination of single-threaded one-rule Semi-Thue systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 338–352. Springer, Heidelberg (2005)
11. Zantema, H.: Termination of term rewriting by semantic labelling. *Fundamenta Informaticae* 24, 89–105 (1995)
12. Zantema, H.: Termination. In: *Term Rewriting Systems*, by Terese, pp. 181–259. Cambridge University Press, Cambridge (2003)
13. Zantema, H.: Termination of string rewriting proved automatically. *Journal of Automated Reasoning* 34, 105–139 (2004)
14. Zantema, H., Geser, A.: A complete characterization of termination of $0^p 1^q \rightarrow 1^r 0^s$. In: Hsiang, J. (ed.) *Rewriting Techniques and Applications*. LNCS, vol. 914, pp. 41–55. Springer, Heidelberg (1995)

Author Index

- Anantharaman, Siva 20
Balland, Emilie 36
Boichut, Yohan 48
Boy de la Tour, Thierry 63
Brauner, Paul 36
Cirstea, Horatiu 78
Dowek, Gilles 93
Durand, Irène 107
Duval, Dominique 122
Echahed, Rachid 122,137
Echenim, Mnacho 63
Escobar, Santiago 153
Espírito Santo, José 169
Fages, François 214
Faure, Germain 78
Genet, Thomas 48
Godoy, Guillem 184, 200
Haemmerlé, Rémy 214
Hendrix, Joe 229
Hermant, Olivier 93
Hills, Mark 246
Huntingford, Eduard 184, 200
Jensen, Thomas 48
Kikuchi, Kentaro 257
Kopetz, Radu 36
Korp, Martin 273
Kutsia, Temur 288
Le Roux, Luka 48
Leroy, Xavier 1
Levy, Jordi 288
Marché, Claude 303
Meseguer, José 153, 229
Middeldorp, Aart 273, 389
Moreau, Pierre-Etienne 36
Narendran, Paliath 20
Nieuwenhuis, Robert 2
Oliveras, Albert 2
Peltier, Nicolas 137
Pfenning, Frank 19
Prost, Frederic 122
Roşu, Grigore 246
Reilles, Antoine 36
Rodríguez-Carbonell, Enric 2
Rubio, Albert 2
Rusinowitch, Michael 20
Schmidt-Schauß, Manfred 329
Sénizergues, Géraud 107
Straßburger, Lutz 344
Tatsuta, Makoto 359
Tiwari, Ashish 200
van Oostrom, Vincent 314
Vaux, Lionel 374
Villaret, Mateu 288
Waldmann, Johannes 404
Zankl, Harald 389
Zantema, Hans 303, 404