24th International Conference on Rewriting Techniques and Applications

RTA'13, June 24–26, 2013, Eindhoven, The Netherlands

Edited by

Femke van Raamsdonk

Editor

Femke van Raamsdonk
Department of Computer Science
VU University
Amsterdam, The Netherlands
f.van.raamsdonk@vu.nl

ACM Classification 1998
D.1 Programming Techniques, D.2 Software Engineering, D.3 Programming Languages, F.1 Computation by Abstract Devices, F.2 Analysis of Algorithms and Problem Complexity, F.3 Logics and Meanings of Programs, F.4 Mathematical Logic and Formal Languages, I.1 Symbolic and Algebraic Manipulation, I.2 Artificial Intelligence

ISBN 978-3-939897-53-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-53-8.

Publication date
June, 2013

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.RTA.2013.i

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

Invited Talks

Regular Papers

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the proceedings of the 24rd International Conference on Rewriting Techniques and Applications (RTA 2013), which was held June 24–26 2013, in Eindhoven, the Netherlands. 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), New Brunswick (1996), Sitges (1997), Tsukuba (1998), Trento (1999), Norwich (2000), Utrecht (2001), Copenhagen (2002), Valencia (2003), Aachen (2004), Nara (2005), Seattle (2006), Paris (2007), Hagenberg/Linz (2008), Brasilia (2009), Edinburgh (2010), Novi Sad (2011), and Nagoya (2012).

RTA 2013 received 50 submissions from 16 countries. The programme committee selected 20 regular papers and 1 system description for presentation at the conference. The selection process greatly benefitted from the conscientious and excellent work of in total 71 external reviewers, some of whom reviewed more than one paper.

The programme committee selected the contributions *Linear Logic and Strong Normalization* by Beniamino Accattoli and *A Homotopical Completion Procedure with Applications to Coherence of Monoids* by Yves Guiraud, Philippe Malbos, and Samuel Mimram together for the best paper award.

Jarkko Kari, Mitsuhiro Okada, and Simon Peyton Jones presented invited talks at RTA 2013. The talk by Simon Peyton Jones was joint with TLCA 2013. It is a great pleasure to thank the invited speakers for enriching the conference with their talks and their participation, and for their contributions to the present proceedings.

We used the EasyChair system for many aspects of the reviewing process. We wish to thank Andrei Voronkov and all others of the EasyChair team for this invaluable tool.

The proceedings of RTA 2013 are published as a volume in the LIPIcs series. I wish to thank the editorial board of LIPIcs for agreeing to publish these proceedings, and the team of the LIPIcs editorial office for their help in the preparation of these proceedings. I thank in particular Marc Herbstritt for always promptly and accurately helping out when needed.

RTA 2013 was organized as part of the International Conference on Rewriting, Deduction, and Programming (RDP 2013). I would like to thank Hans Zantema, the conference chair of RTA 2013, and the other members of the organizing committee of RDP 2013 for their indispensable contribution to the success of RDP 2013. RDP 2013 accommodated also the 11th International Conference on Typed Lambda Calculi and Applications (TLCA 2013). It was a pleasure to work together with Masahito Hasegawa, the programme chair of TLCA 2013. Further, several workshops were organized, including the Annual Meeting of the IFIP Working Group 1.6 on Term Rewriting, the Workshop on Haskell and Rewriting Techniques (HART 2013), the 27th International Workshop on Unification (UNIF 2013), the 2nd International Workshop on Confluence (IWC 2013), and the Workshop on Infinitary Rewriting (WIR 2013). I thank the organizers of all workshops for making the programme of RDP 2013 more diverse and attractive.

I am very grateful to all members of the RTA 2013 programme committee for their reviews, their constructive comments, and the pleasant collaboration. A special word of thanks to Aart Middeldorp and Vincent van Oostrom for their asked and unasked advice.

RTA 2013 gratefully acknowledges the financial support of NWO (the Netherlands Organisation for Scientific Research) and the support of Eindhoven University of Technology.

June 2013 Femke van Raamsdonk

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conference Organization

Programme Chair

Femke van Raamsdonk	VU University Amsterdam

Conference Chair

Hans Zantema	Eindhoven University of Technology

Programme Committee

Eduardo Bonelli	National University of Quilmes
Byron Cook	Microsoft Research Cambridge
Stephanie Delaune	ENS Cachan
Gilles Dowek	Inria Paris–Rocquencourt
Maribel Fernández	King's College London
Nao Hirokawa	JAIST Ishikawa
Delia Kesner	University Paris-Diderot
Hélène Kirchner	Inria Paris–Rocquencourt
Barbara König	University Duisburg Essen
Temur Kutsia	Johannes Kepler University Linz
Aart Middeldorp	University of Innsbruck
Vincent van Oostrom	Utrecht University
Femke van Raamsdonk	VU University Amsterdam
Kristoffer Rose	IBM Research New York
Manfred Schmidt-Schauß	Goethe University Frankfurt
Peter Selinger	Dalhousie University
Paula Severi	University of Leicester
Aaron Stump	The University of Iowa
Tarmo Uustalu	Institute of Cybernetics Tallinn
Roel de Vrijer	VU University Amsterdam
Johannes Waldmann	HTWK Leipzig
Hans Zantema	Eindhoven University of Technology

Steering Committee

Mauricio Ayala-Rincón	Brasilia University
Frédéric Blanqui	INRIA Tsinghua University Beijing
Salvador Lucas	Technical University of Valencia
Georg Moser (chair)	University of Innsbruck
Masahiko Sakai	Nagoya University
Sophie Tison	University of Lille

RTA Web Page

http://rewriting.loria.fr/rta/

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Accattoli, Beniamino
Albert, Elvira
Andrei, Oana
Aoto, Takahito
Ayala-Rincón, Mauricio

Baelde, David
Balabonski, Thibaut
Bau, Alexander
Blom, Stefan
Brockschmidt, Marc
Bruggink, H.J. Sander
Bucciarelli, Antonio

Chevalier, Yannick
Cirstea, Horatiu
Conchon, Sylvain
Courtieu, Pierre

Eades, Harley
Echahed, Rachid
Eker, Steven

Falke, Stephan
Felgenhauer, Bertram
Fouquere, Christophe
Fu, Peng
Fuhs, Carsten

Gascón, Adrià
Genet, Thomas
Gimenez, Stéphane
Grabmayer, Clemens
Graham-Lengrand, Stéphane
Gust, Helmar

Habermehl, Peter
Heckel, Reiko
Hirschkoff, Daniel
Hofbauer, Dieter

Jeltsch, Wolfgang

Kapur, Deepak
Ketema, Jeroen
Klop, Jan Willem
Kop, Cynthia

Levy, Jordi
Licata, Daniel R.
Lushman, Brad

Madet, Antoine
Marin, Mircea
Matsuda, Kazutaka
Mazza, Damiano
Minas, Mark

Nakata, Keiko
Nakazawa, Koji

Otto, Carsten

Padovani, Vincent
Payet, Etienne
Petrişan, Daniela
Plump, Detlef

Rau, Conrad
Rémy, Didier
Rochel, Jan
Rose, Eva

Sabel, David
Schaper, Michael
Schneider-Kamp, Peter
Sternagel, Christian
Straßburger, Lutz
Stückrath, Jan

Thiemann, René
Toyama, Yoshihito

Urban, Christian

Vries, Fer-Jan de

Wiedijk, Freek
Winkler, Sarah

Zankl, Harald

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Author Index

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Pattern Generation by Cellular Automata* (Invited Talk)

Jarkko Kari

Department of Mathematics and Statistics
FI-20014 University of Turku, Finland
jkari@utu.fi

Abstract

A one-dimensional cellular automaton is a discrete dynamical system where a sequence of symbols evolves synchronously according to a local update rule. We discuss simple update rules that make the automaton perform multiplications of numbers by a constant. If the constant and the number base are selected suitably the automaton becomes a universal pattern generator: all finite strings over its state alphabet appear from a finite seed. In particular we consider the automata that multiply by constants 3 and 3/2 in base 6. We discuss the connections of these automata to some difficult open questions in number theory, and we pose several further questions concerning pattern generation in cellular automata.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases cellular automata, pattern generation, Z-numbers

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.1

1 Enumerating all patterns by a cellular automaton

Cellular Automata (CA) are parallel and synchronous rewrite systems with local dependencies. The system consists of a regular grid of *cells*, each storing a single symbol called the *state* of the cell. The cells change their states synchronously according to a *local update rule* that specifies the new state depending on the local pattern of states around the cell. As cellular automata obey fundamental principles of physics such as locality and uniformity in space and time, they have found applications in various modeling situations of natural systems [2].

Cellular automata were first introduced by John von Neumann, following a suggestion by Stanislaw Ulam, to demonstrate an abstract universal constructor in an artificial setting [9]. Since then, the complexity that can arise from simple local rules and simple seed patterns has been demonstrated several times. Most notably, the well-known *Game-of-Life* cellular automaton by John Conway supports universal computation [1], as does *Rule 110*, a one-dimensional cellular automaton with binary alphabet and radius-one local update rule [3].

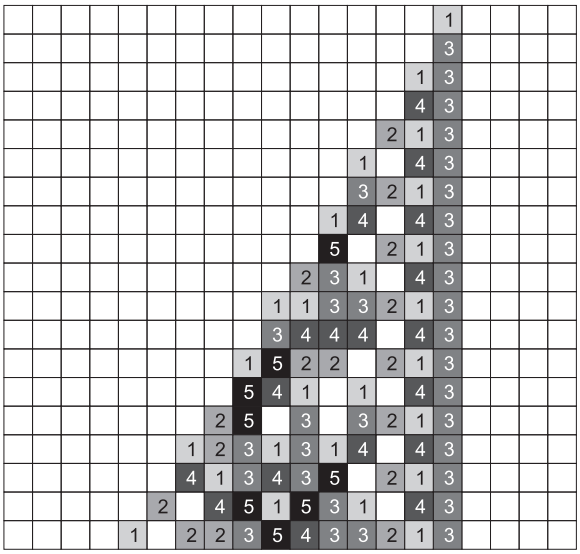
In this talk we consider a question asked by Stanislaw Ulam about generating all patterns from a single finite seed [8, page 30]. The problem is to design a cellular automaton rule and an initial configuration with all but finitely many cells in null states such that in the evolution that follows all finite patterns over the state alphabet will appear. We use two simple facts to design such a rule [6]: (i) the powers of a number n written in base b contain all finite digit sequences if n is not a rational power of b, and (ii) the multiplication of numbers by n in base b is a cellular automaton if all prime factors of n also divide b. Smallest such example

* Research supported by the Academy of Finland Grant 131558.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 1–3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


is the cellular automaton $F_{\times 3}$ that multiplies by $n = 3$ in base $b = 6$. The evolution from a single digit 1 looks as follows:



The automaton is time reversible, which means that another cellular automaton traverses the same configurations backwards in time. The existence of this simple solution raises other interesting questions to investigate:

- Do there exist analogous universal pattern generators also in two- and higher dimensional cellular spaces ?
- Does there exist a solution with fewer than six states ? In particular: is there a universal pattern generator over the binary alphabet ?
- Does there exist a solution that generates all patterns at all positions ? In terms of the usual product topology on the configuration space: Does there exist a cellular automaton with a dense orbit of finite configurations ?

2 Number theoretic questions

A candidate to solve the last problem is obtained by combining $F_{\times 3}$ with a suitable right shift. This suggests the automaton $F_{\times 3/2}$ that multiplies numbers in base 6 by constant $3/2$. The problem arises to determine if all sequences of digits get generated next to the radix point, i.e., whether the fractional parts of the powers of $3/2$ are dense in the interval $[0, 1]$. More precisely, one needs to find an integer m such that the fractional parts of $\xi(3/2)^i$ are dense for all $\xi = m/6^k$.

The automaton that multiplies by $3/2$ relates also to other number theoretic problems in a natural way [5]. In [7], a *Z-number* was defined to be any real $\xi > 0$ with the property that the fractional part of $\xi(3/2)^i$ is less than one half for all $i = 0, 1, 2 \ldots$. The problem of whether any Z-numbers exist is still unsolved. If $\xi(3/2)^i$ is written in base 6, the requirement is simply that the first digit after the radix point is 0, 1 or 2 (with the minor exception that the fractional part may not be $.2555\ldots$). Hence the existence of Z-numbers can be rephrased as a question concerning time evolutions by $F_{\times 3/2}$ at a single site.

Finally, by adding a new state to represent a floating radix point, we modify $F_{\times 3}$ to

simulate the Collatz-function

$$m \mapsto \begin{cases} m/2, & m \text{ even}, \\ 3m+1, & m \text{ odd}, \end{cases}$$

on base 6 representations of positive integers [5].

References

1 E.R. Berlekamp, J.H. Conway, and R.K. Guy. *Winning ways for your mathematical plays, Volume 2: Games in Particular*. Academic Press, 1982.

2 B. Chopard and M. Droz. *Cellular automata modeling of physical systems*. Cambridge University Press, 1998.

3 M. Cook. Universality in Elementary Cellular Automata. *Complex Systems*, 15(1):1–40, 2004.

4 J. Kari. Theory of cellular automata: A survey. *Theor. Comput. Sci.*, 334(1-3):3–33, 2005.

5 J. Kari. Cellular automata, the collatz conjecture and powers of 3/2. In H.-C. Yen and O. Ibarra, editors, *Developments in Language Theory*, volume 7410 of *Lecture Notes in Computer Science*, pages 40–49. Springer, 2012.

6 J. Kari. Universal pattern generation by cellular automata. *Theoretical Computer Science*, 429(0):180 – 184, 2012.

7 K. Mahler. An unsolved problem on the powers of 3/2. *Journal of The Australian Mathematical Society*, 8:313–321, 1968.

8 S.M. Ulam. *A Collection of Mathematical Problems*. Interscience, New York, NY, USA, 1960.

9 J. von Neuman. *The Theory of Self–Replicating Automata (Ed. Burks, A. W.)*. University of Illinois Press, Urbana, IL, 1966.

Husserl and Hilbert on Completeness and Husserl's Term Rewrite-based Theory of Multiplicity (Invited Talk)

Mitsuhiro Okada

Department of Philosophy, Keio University
2-15-45 Mita, Minato-ku, Tokyo, Japan
mitsu@abelard.flet.keio.ac.jp

Abstract

Hilbert and Husserl presented axiomatic arithmetic theories in different ways and proposed two different notions of "completeness" for arithmetic, at the turning of the 20th Century (1900–1901). The former led to the completion axiom, the latter completion of rewriting. We look into the latter in comparison with the former. The key notion to understand the latter is the notion of *definite multiplicity* or *manifold* (Mannigfaltigkeit). We show that his notion of multiplicity is understood by means of term rewrite theory in a very coherent manner, and that his notion of "definite" multiplicity is understood as the relational web (or tissue) structure, the core part of which is a "convergent" term rewrite proof structure. We examine how Husserl introduced his term rewrite theory in 1901 in the context of a controversy with Hilbert on the notion of completeness, and in the context of solving the justification problem of the use of imaginaries in mathematics, which was an important issue in the foundations of mathematics in the period.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases History of term rewrite theory, Husserl, Hilbert, proof theory, Knuth-Bendix completion

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.4

1 Introduction

Two characteristic notions of completeness of arithmetic appeared at the same place, Göttingen in Germany, in 1900–1901, at the same Faculty of Philosophy, one introduced by Hilbert of the Mathematics Section, and the other introduced by Husserl of the Philosophy Section. The notion of completeness by Hilbert is well known: completeness in his sense ensures existence of a categorical model of an axiomatic system. On the other hand, Husserl's notion of completeness is not well known: completeness in his sense ensures a mathematical multiplicity or manifold (Mannigfaltigkeit) to be "definite". His notion of definite multiplicity has not been clarified very well until today although many efforts for clarifications have been made by a large number of former works. The purpose of this paper is to show that Husserl's definite multiplicity, hence completeness, can be well understood by the helps of term rewrite theory.

We explain how Husserl's notion of completeness is different from Hilbert and how he reached his idea by interpreting Hilbert's axiomatization of arithmetic in a slightly different way. We, in particular, show that Husserl introduced the various basic notions of term rewrite proofs. His motivation of the study on completeness was originated from "the justification problem of the use of imaginaries in mathematics", which is concerned with the conservation problem in the modern logical sense. He reached his solution in 1901 and presented it at

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 4–19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


the Mathematical Society Meeting invited and organized by Hilbert at Göttingen in the same year. The lectures is now called his Double Lecture (November and December). as he gave two talks there. He gave a solution using the notions of syntactic completeness and consistency (syntactic completeness for the original system and consistency for enlarged system with imaginaries) as the first rough outline. However he went further to clarify as to what was the completeness condition, then he explained it by means of, essentially, term rewrite theory. Husserl introduced his notion of definiteness of multiplicity or manifold to explain his notion of completeness. We show that a multiplicity of an axiomatic system, in his sense, is the whole web (or tissue) structure of rewrite equational-proof formation steps (or reduction moves) and term-formation steps (or moves), and that the notion of definiteness of a multiplicity corresponds to the notion of convergence in the modern term rewrite theory. A form of (Knuth-Bendix) completion procedure was also proposed by Husserl in 1901, in order to make a non-definite multiplicity definite. We shall explain these with some textual evidences (with the use of Husserl's several manuscripts in 1901); more detailed references and quotations as well as more philosophical discussions of the subject will appear in the subsequent papers. Husserl in fact used the word "term" and considered the notion of multiplicity as the term-rewrite computational content of an axiomatic system. But he also used the words, "concept" (of a term), and "object of of a concept" (of term), keeping his philosophical framework of significative intention-objectivity in the Logical Investigations. Moreover, the term-rewrite reductions to normal forms was considered as the significative fulfillment framework of the Sixth Investigation, to some extent. We shall discuss the further philosophical discussions related to these subjects in the subsequent philosophical paper.[1]

Roughly speaking, by completeness of arithmetic Hilbert meant the maximal extension of a model to reach a categorical (unique) model, which is the continuum, while Husserl meant the minimal term model based-reductive proofs web (tissue) structure, which is, roughly speaking, his notion of definite multiplicity. Husserl argued, against Hilbert, that this way to understand completeness is needed to solve the problem of justifying the use of imaginaries in mathematics.[2] Husserl distinguished the two notions of completeness clearly from the point of view of the justification problem of imaginaries, as if the axiomatic system were complete in the Hilbert sense (which he called "essentially complete" compared with his "outer-essentially complete"), the purpose of him to investigate in the problem would be hidden since no possibility of extending the original complete axiomatic system to an enlarged system with imaginearies would remain[*Hua XII*, p.445] . Hence, in order to understand the background issues and the motivation how Husserl reached a theory of rewriting proofs, we explain the problem of justifying the use of imaginaries in mathematics briefly. The problem of justification of the imaginaries had been one of the issues in philosophy of mathematics since Leibniz in the 17th Century, but in particular in the period of the end of the 19th Century. Husserl reached in 1901 to a general theory of convergent term rewriting and completion in the context of attacking this question.

The question of justifying the use of imaginaries in mathematics had been raised by various scholars both in mathematics and in philosophy at the turning of the century, and

[1] In this paper, we do not compare our work with former works related to the Double Lecture, except for a limited number of references and comments. We shall discuss further comparisions in the subsequent paper.

[2] It is also noted that Hilbert first considered an axiomatic foundations of geometry and reduced his foundational issues on consistency of geometry to that of (real number) arithmetic, while Husserl did in the opposite way; he believed his term rewrite theory on arithmetic should work with geometry.

the research question was in the air.[3] A typical way of the questioning was: "How can we justify the use of imaginary numbers in the course of proving a proposition or calculating an equation referring only to concepts on real numbers? Can such a proof or calculation through imaginaries be always transformed to a proof or calculation referring only to concepts on real numbers?"

Another example would be: "How can we justify an analytic proof techniques for proving of a theorem on elementary number theory?"

Husserl considered various cases of a relatively real system and its enlarged system with relatively imaginaries with respect to the real, and attacked the problem of imaginaries in a general arithmetical setting in 1901, to tried to find a general condition (from the term-rewrite theoretic view) to settle the problem.[4] Hilbert in his peak period of logic-foundational studies (in the 1920's) also considered a general framework of enlarged system with imaginaries, although Hilbert stuck to a fixed "contentual" finitist mathematical system for the original "real" system.[5]

The problem of justification of imaginaries in mathematics in the Husserl-Hilbert style is understood (at least roughly). as the problem to show (a sort of) "conservative relation" of axiomatic systems. Consider two formal axiomatic systems $S_1 \subset S_2$ where

S_1: original [real] system,
S_2: enlarged system with some imaginaries

[Conservation Property] For any real proposition A of S_1, if A is provable in S_2 then A is always provable in S_1, namely,

$$S_2 \vdash A \Longrightarrow S_1 \vdash A,$$

then S_2 is called a conservative extension of S_1 (and S_1 is called a conservative subsystem of S_2)[6]

---

[3] See [Hartimo 2007] [Hartimo 2010] for the historical background and context at the end of the Century on the problem of justification of the use of imaginaries around the turning of the century.

[4] They include any variant of the usual formal axiomatic natural number theory, as well as any formal theory of integers, of rationals, of real numbers, of complex numbers, etc.[Schuhmann and Schuhmann 2001, p.105–p.106], [*Hua XII*, p.442–p.443] as well as (propositional) logical calculus [*Hua XII*, p.487–p.488]

[5] Cf. [Hilbert 1926], also [Detlefsen 1947] and [Kreisel 1958], for Hilbert's consistency proof program. It is possible that Hilbert's (revised) presentation of the consistency program in the mid 1920's, using the conservation problem framework, was a result of the influence of Husserl's 1901 talk to Hilbert, which was pointed out in [Okada 1987].

[6] The conservative delation problems, hence the justification problems, had been discussed in history of the development of mathematics (although the property cannot always be expected as Gödel showed in his Incompleteness Theorem (1931). Note that both Leibniz and Gödel emphasizes the usefulness of introducing imaginaries by pointing out the effect of speeding-up and servayability of proving the real propositions. Leibniz, pointed out usefulness of placing Lemma (hence the use of cut-rule in the sense of Gentzen, while the investigation into rewriting of a proof with cut-rules (hence with lemmas) into a cut-free direct proof was one of the main research paradigms employed by the Hilbert School in the 1030's in order to solve the conservation-justification problem. In the case of Leibniz, he needed the use of infinitesimal (real) numbers, such as dx, and infinite (real) numbers, $1/dx$ in his introduction of differential and integral calculus in the 17th Century. Of course, he himself faced the question how to justify the use of such numbers. Now, there are two well known ways to justify the use; one is to introduce the "contextual" rewriting with of "limit"; for example, $df(x)/dx$ is defined contextually with the ϵ-δ "description", which was introduced by Cauchy, only in the beginning of the 19th Century, then more logically by Bolzano and Weierstrass (Husserl worked as an assistant to Weierstrass before he moved to philosophy.). The other way is to introduce nonstandard analysis; Abraham Robinson, in the 1950-60's, was first who adapted Tarskian model theory/formal semantics theory to Leibniz representation dx as the nonstandard real numbers in his introduction of non-standard analysis, where the notion of elementary extension of a model is essential. Leibniz himself also suggested the first way, but also expressed that a certain algebraic or abstract rewriting works in practice (cf. [Okada 1987])

Husserl prepared various manuscripts in the winter semester of 1901, and presented his "solution" to the problem, which he believed to work for various arithmetic systems at two successive talks, which is now called the Double Lecture, at the Göttingen Mathematical Society Meeting organized by Hilbert. At the first talk he claimed, among others, the following.

(Claim α) If the following are satisfied, then the use of imaginaries is justified.

(1) the original narrower system is (syntactically) complete, and
(2) the enlarged system is consistent.

Namely, the above two conditions imply the conservation property i.e., the enlarged system is a conservative extension over the original, hence the use of imaginaries introduced in an enlarged system is justified under these conditions.[7]

He explained his notion of completeness more precisely by the use of the notion of multiplicity or manifold (Mannigfaltigkeit). The notion of completeness is characterized by "definiteness" of a multiplicity. The word "multiplicity" of an axiomatic system and the word "domain of an axiomatic system" were exchangeably used by him.[8] The originality of this paper is to clarify (or propose to read) the notion of definite multiplicity of Husserl in terms of term rewrite theory. In fact, Husserl went beyond just the notion of syntactic completeness by going into the notion of (definite) multiplicity. Since his notion of enlargement of definite multiplicity guarantees consistency[9], he expresses his claim as follows.

(Claim β) An axiomatic theory is complete "If the [an] axiomatic mathematical theory determines its mathematical domain (multiplicity) "definitely," without leaving any ambiguity in the structure of the "domain", and then the use of imaginaries is justified".

Husserl called a multiplicity which is determined definitely a definite multiplicity.

Before we go to the next Section, for philosophical readers we make here a remark; Husserl reached his notion of definite multiplicity with term rewriting theory as the result of his various different former studies of him. The following different questions and studies merged at the same time in 1901 winter when he reached the solution.

1. justification of the use of imaginaries in mathematics,
2. mathematical multiplicity/manifold,
3. general conditions of extending/overloading mathematical operators/functions (consistent overloading use of function symbols), or now called of overloading (of function symbols) in Computer Science,
4. general theory of decision problem for an axiomatic equational system.
5. phenomenological notion of significative fulfillment as a fulfillment of arithmetic terms .
6. studies in categorial intuition (or syntax-oriented intuitive evidence, including intuition on formal arithmetic).

---

[7] This was pointed out in [Okada 1987] and [Majer 1997].
[8] This fact might have misled the commentators to understand the notion as a set theoretical model of the axiomatic system for the long time history of study over 60 years.
[9] as we shall see later, he considered extending an original system under the condition that normal (irreducible) constructor terms do not collapse

Unfortunately in this short paper we cannot explain and discuss the whole picture of how Husserl united 1-6 above during his development in his philosophy of logic and mathematics. We would like to pick up some minimum backgrounds and issues from these lists, and discuss a birth of an ideas of definite multiplicity and rewriting theory by Husserl. We leave more detailed discussions on the controversy between Hilbert and husserl, from the philosophical and mathematical points of view in the subsequent papers.

2 Hilbert's axiomatic system of arithmetic and Husserl's interpretation of Hilbert's system toward term rewrite theory

Hilbert published in 1900 his article on his formulation of arithmetic axioms in 1900 (in the same line as his "Foundations of geometry (1899)". Husserl modified Hilbert's axiomatization of arithmetic in 1901, which shows us some important differences between the attitudes of the two figures on the notion of completeness.[10] Husserl presented a formal axiomatic system in a form very similar to Hilbert's, with a slight modification. Here, this similarity and slight modification are both important, in our opinion, to understand Husserl's notion of completeness, and of his term rewrite based-notion of multiplicity.

Hilbert's presentation of axioms for arithmetic in question is composed of four groups of axioms, as well known:

I. Axioms of linking (junction)
II. Axioms of calculation
III. Axioms of ordering
IV. Axioms of continuity (composed of the Archimedean axiom and the axiom of completeness (closure) saying that the model of the axiomatic system is categorical, in the sense that the maximal closure of the models is unique..)

Now we go through to check Husserl's modified understanding of Hilbert's axiomatic system of arithmetic.

Axioms of Continuity The Group IV (the continuity) is composed of the Archimedean axiom (IV-1) and the axiom of completeness (IV-2). The axiom of completeness says the that the only maximally extended model (unique up to isomorphism) of the models of axiomatic system of (I)-(IV-1) is the model of the whole system (I)-(IV).[11]

Although this completeness axiom is placed to intend to give the unique determination of semantical model-structure by mean of a syntactic axioms„ the expression of the completeness

[10] Husserl's analysis on Hilbert's axioms of arithmetic appeared only in Schumann-Schumman's edition [Schuhmann and Schuhmann 2001] of the Double Lecture manuscript, not in the original edition[*Hua XII*]. It is plausible that this part (Husserl's critical modification of Hilbert's axiomatic system) was written by Husserl only after his first lecture of the Double-Lecture after the discussion with Hilbert, where Hilbert was among the audience. Hilbert had already published "on the number concept" in 1900 [Hilbert 1900] in which he presented formal axiomatic system for arithmetic with his notion of completion.

[11] Hilbert expresses categoricity of the model of its own axiomatic system. Hilbert expressed the axiom of completeness as follows in [Hilbert 1900] [Ewald 1996], He wrote:

> It is not possible to add to the system of numbers another system of things so that the axioms I, II, III and IV-1 [namely, all the axioms except this completeness axiom itself] are also all satisfied in the combined system; in short, the numbers form a system of things which is incapable of being extended while continuing to satisfy all the axioms.

Here, a "system of things" means a "model" in the contemporary logical sense.

axiom itself refers to the semantic notion. Hence, it cannot be understood in the framework of the contemporary syntax-semantics distinction. Husserl proposed to stay on the syntactic side to express axioms. This is the starting point of Husserl's way of considering his notion of completeness.

Hence, Husserl, of course, abandoned the axioms Group IV, i.e., axioms of continuity. He explained before Hilbert in the audience that the Hilbertian completeness axiom excludes possibility of extending axiomatic system with imaginaries because the completeness axiom of Hilbert requires maximality (non-extendability) of the model„ (hence excludes the question of imaginaries itself under this axiomatic setting.[12]

Axioms of Linking The linking axioms (Group I) in both Hilbert's and Husserl's formation state that the primitive operation symbols (function-symbols) carry out a linking among the terms on the term formations (generations). Namely, when "$+$" ("$\times$", respectively) is used with terms, say s and t, a new term $s+t$ ($s \times t$) is formed: $+$ links the two terms s and t to the new composed terms $s+t$, and $s \times t$. Although Husserl's presentation of the axiomatic system of arithmetic (he presented, as an example, system of rational number) was surprisingly similar to Hilbert's there is an important difference between them on this Linking Axioms. For Hilbert the new linkage $s+t$ (or *stimest*) gives a "determinant" number. On the other hand, Husserl was concerned with possibility of indeterminacy with concrete presence of calculation axioms,. In fact, he was concerned with non-confluent rewrite calculations and non-terminating rewrite calculation. For him, determinacy of terms needs to be characterized by means of the term rewrite structure imposed by the Calculation Axioms, which is the basis of his notion of multiplicity.[13]

In the case of Husserl, for a term, say t (in Husserl's terminology, operational complexity), "t" is called "provably existent" when t is reduced to be a constructor-based normal term, say n, with the help of the axioms of calculation. Husserl's completeness and definiteness of multiplicity means, as we shall see in next Section, that this linking edges for the term-formations are most compactly, hence minimally determined, in accordance with the minimal term-model, while Hilbert's completeness or closure axiom means that the linkages are fixed in the maximally expanded way in the sense of the categorical model.[14]

Axioms of Ordering This part is the same as that of Hilbert.[15]

Axioms of Calculation Now, Group II, in which an outstanding difference can be found, as Husserl needs to claim that an axiomatic system forms the proof structure of definite multiplicity, which requires at least a ground convergent term rewrite proof structure (as we see in the next Section more closely). for which he understand that under the setting of the calculation axioms any closed term should have rewriting deductive steps i.e., to a unique

[12] This explanation was put just before the main part of his completeness proof of arithmetic in the Double Lecture manuscript.

[13] We use the English words "junction" and "linking" interchangeably for the translation of "Verknüpfung." On the other hand, when Husserl describes the term-rewrite based-reduction structure of a multiplicity he uses the word "term" (Glied).

[14] The axioms of linking in Hilbert include not only the term formation definition but also the characterization of idempotent-functions and converse-functions, as he consider fields. Husserl employed this Hilbert line to define constructors, $0, 1$ following Hilbert, for his notion of constructive multiplicity. see below..

[15] Group III, the ordering axioms of the ordered field, is exactly the same for both Husserl and Hilbert although Husserl writes down precisely only a few examples of Hilbert's full axioms.

normal term, (a unique term representing a rational number as he uses the system of rational numbers as an example at the Double Lecture). It is particularly interesting to see that four pages earlier in the (Schuhmann-Schuhmann 2001) edition of the same manuscript, Husserl tried to formulate the axiomatic systems and to start with an Hilbertian axiom of calculation (Group II); there he put first the commutativity axiom for "+" (i.e., $a + b = b + a$),[16] which is one of the six axioms of Hilbert's Group II (Hilbert's six axioms are the commutativity, associativity and distributivity for the two primitive function symbols + and ×.), in the later part of the Double Lecture manuscript, Husserl did not mention this commutativity rule to discuss Group II axioms. It means, in the author's opinion, that Husserl intended to change the form of Calculation Axioms (Group II) to make the rewrite rules oriented, by changing the algebraic rules to the rewrite rules. He presented a reductive rewrite evaluations in the 6th Logical Investigation in 1900 (Section 60) , (in his phenomenological terminology, "significative fulfillment"). (In the Double Lecture and other manuscripts in 1901 which we mentioned rarely used the phenomenological vocabulary but still he used the words "fulfillment" and "adequation".)

He did not present concrete list of rewrite rules as the calculation axioms, but his way was to present a general term rewrite theory in the sense that what kind of condition the rewrite rules should satisfy in order to the axiomatic system convergent, hence complete.[17]

Although it is not very clear what are the exact form of new calculation rules of Husserl, in any case, it is very clear that Husserl stepped out of Hilbert's setting of calculation rules here and realized the need for completely different axiomatizations of equational calculation rules to govern calculation of each operation (function) represented by a function symbol, which shows that each one-step move from one joint to another on the joint-web (tissue) structure of multiplicity, which can be performed by means of equational deduction (namely, a particular application of an algebraic general axiom(s) of calculation needs to correspond to the underlying one step ground-term rewriting).

We might need too point out here that the explicit primitive recursive calculation axioms were presented only more than 20 years later by Skolem. In particular he also presented (in an informal way) the mathematical induction scheme in his Primitive Recursive Arithmetic (PRA).

3 Husserl's definite multiplicity as the convergent term rewriting proofs web

Now we have reached the stage to discuss Husserl's notion of definite multiplicity in the Double Lecture and other important and matured manuscripts in 1901 and to explain how the notion is directly related to term rewrite theory.

We first explain the notion of multiplicity (manifold) of an axiomatic system in Husserl's sense. Husserl also uses the word "domain of an axiomatic system" to express a multiplicity. Husserl's multiplicity has been interpreted by many philosophers and logicians for over 40 years that this domain-multiplicity means a set theoretical domain, namely a model in the sense of model theory.[18] We show now that this is not the case by our reading and that by a multiplicity he means the whole network (web or tissue) of rewrite equational proofs and term formations.[19]

---

[16] [Schuhmann and Schuhmann 2001, p.113]

[17] In fact, Husserl allowed to introduce any number of function symbols in Group I (Linkage Axiom), hence it is natural to presume that he consider calculation axioms to calculate those function symbols.

[18] Some other views may be found in [Hartimo 2010]

[19] In fact, enlarging an original axiomatic system by adding imaginary propositions as new axioms on the

(A) Multiplicity Husserl's multiplicity of an axiomatic system is understood as the following whole web (or tissue) or graph structure (relation-web or relation-linkings):

(a) the nodes represented by terms of the system, and the edges represented by following relations between terms;
(b) term formation steps following the linking axioms,
(c) rewrite (i.e., oriented equational) proof steps following the calculation axioms.
(d) provably decidable atomic relations of normal (irreducible) terms in the rewrite proof sense.

A multiplicity is said to be definite if the web or graph is tightly determined. In particular he requires convergence of the rewrite edges.

(B) Definite Multiplicity A multiplicity is called definite (in the strong sense) when the underlying term rewrite system is convergent. (confluent and terminating).

There is no word "convergence" nor other words which are used in term rewrite theory, but by definiteness he meant the direction-oriented equational deductions confluent and terminating.

He often confirms that an arithmetical multiplicity is ground convergent (namely, convergent on the ground (closed) terms level, and left open the convergence on the variables level, and he believes that the calculation axioms could set so that the convergence on the ground level holds. In this paper we distinguish the definite only on the ground convergence case from the general case.

(B') Definite Multiplicity in the weak sense A multiplicity is called definite in the weak sense when the underlying term rewrite system is ground convergent. (confluent and terminating on the closed terms level).

Husserl also defines constructiveness of the definite multiplicity, where the constructor terms are pre-given, namely formed by linking axioms based on given constructors.

(C) Constructive Multiplicity When the term formation steps are based with constructors, and the calculation axioms preserve the constructor terms as (at least a part of) the normal (irreducible terms), the whole web (or tissue) is "constructive" multiplicity[20].

He focuses the notion of constructive multiplicity especially in the Double Lecture. He presumes that the constructor terms are irreducible in the sense that they are normal terms. The termination property comes with this setting. He calls the pre-given distinguished set of intended normal terms as the number-series. He follows Frege and Hilbert regarding this naming. He, however, also calls the number-series as the "standard" or "measure" . He gives his dynamic term-rewriting view that the measure plays the role of measuring any term in the multiplicity-web (as its value) under the definiteness condition. He imposes a completion procedure on a non-confluent constructive multiplicity by adding new direction-oriented calculation axioms upon necessity so that the resulting multiplicity becomes "definite"

one hand and by enlarging a domain of the original axiomatic system by adding imaginary elements are not equivalent when one assumes the "domain" (or "multiplicity") in the sense of a set theoretical domain and this difficulty is discussed by Husserl himself. Our reading of the domain (multiplicity) as the whole proofs and terms web makes sense and works well by understanding with term rewrite theory in our opinion.

[20] He also uses the word "mathematical multiplicity".

constructive multiplicity; we shall discuss this (Knuth-Bendix type) completion procedure of Husserl later. Husserl claims that

(Claim γ) Rewrite-provability in the sense of direct rewrite proof [without the use of symmetric axiom] is logically equivalent to logical equational provability of equational proof system [with the use of symmetric axiom] in the case that the multiplicity is definite.

This lemma is of course an essential lemma well known in nowadays term rewrite theory. But, to be honest, we should point out that Husserl does not (or could not) treat the termination property directly and explicitly. He rather claims that for a constructive multiplicity case the pre-given set of constructor terms plays as the irreducible terms, and any calculation axioms needs to preserve the irreducibility of the constructor terms (although he allows possibility of reduction paths not falling into the pre-given constructor terms. Therefore, he needs to consider a completion procedure.

Husserl's solution to the problem of justification of the use of imaginaries is expressed as follows.

(Claim δ [Husserl's Solution with the definiteness-completeness condition] If the multiplicity of the original system is definite and the multiplicity of the enlarged system preserves the normal forms of the original, the problem (conservation problem) is positively solved.

Here, we resume to classify the types of multiplicity for an axiomatic formal (deductive) system,according to Husserl:

(i) a (not necessarily definite) multiplicity
(ii) a definite multiplicity
(iii) a constructive (or mathematical) (not necessarily definite) multiplicity
(iv) a constructive definite multiplicity

About Completion from non-definite into definite multiplicity As mentioned above, he also introduces a completion procedure to make the calculation axiom convergent (hence the multiplicity definite) by adding calculation rules for a disjoint (critical) pair. He seems that he was too optimistic about the termination property with respect to this completion procedure as he mainly considers the case of constructive multiplicities where the intended normal terms are pre-given and that the new rules between critical pair can be directed into a constructor normal term side. He call a "disjunctive" moves for the pair of terms which cause non-confluence (hence goes to different irreducible terms.[21] Hence, he claims as follows.

(Claim η) By adding rules for disjoint moves from a position non-convergent constructive multiplicity becomes definite (in a finite steps).

Hierarchical theory of multiplicities, or a part of Mathesis Universalis Husserl often requires definiteness to the multiplicity of enlarged system too, in addition to the preservation of the original normal terms. This is because he considers hierarchical extensions freely by enlarging systems step by step. The accumulated whole is called theory of multiplicities or theory of theories.[22]

---

[21] See [Dershowitz and Jouannaud 1990] for the basic definition of critical pair.

[22] His concrete example of hierarchical theory of multiplicities includes positive number system up to the complex number system as well as logic and geometrical systems), as he presented them concretely in the Double Lecture and related manuscripts.

This was the first (and last) concrete presentation of Husserl's idea of Mathesis Universalis, which he emphasized to aim at establishment in his logical investigations , at the end of his Vol. 1 of the Logical Investigations (Prolegomena), where he also mentioned the problem of justification of the use of imaginaries as an important but had not yet solved the problem.

Husserl describes a multiplicity as the whole *relations-web (or relations-tissue)* of an axiomatic system.[23] "Should a system of axioms define its objects by a *web of relations* (or the form of such a web)" further only by means of materialization, [which means substitutions], "every object must be unambiguously [uniquely] determined by its interrelations [i.e.,by the relation-web]". Then, he continues:

> *Any object is formally the simple position in the relation-web*, i.e., in the relation-form where the objects can be situated, and the form of relation must be so well established that it must be so well formally differentiated in an ultimate manner. If it leaves here indeterminations, it would then again possible to go further in the formal characterization of the *relation-web (tissue)*.[24]

This is a remarkable comment of Husserl on the term rewrite proofs; he says that any (mathematical object of) term in the equational proof system is considered a "simple position" in the whole rewrite-relation-web structure, namely multiplicity, where the objects are situated.[25]

Our "Figure" illustrates an image of a constructive multiplicity web for a simple primitive recursive (hence non-overlapping) calculation axioms just for "+" and "×" where there is the "standard' (or sometimes called "Kernel") ', which is the series of normal terms of numbers[26].

It is definite on the ground level as well as the variable rewrite level, the logical level requires non-equational rules for the ground level though (see below). the multiplicity web of the natural number system is open to expand to various enlarged systems(of integers, of rationals, of computable reals, etc.) with preserving conservation on the ground level.[27]

---

[23] E.g., [*Hua XII*, p.474, 475] (Husserl uses the word "joints (or terms) [Glieder]" instead of "web" in the double-lecture to express it.

[24] [*Hua XII*, p.475]. The author had no chance to carefully look at the English edition of this volume translated by Prof. Dallas Willard during the preparation of this paper. The author plans to consult and quote Prof. Willard's English edition for the preparation of his forthcoming paper on this subject. The author just points out here that the English edition uses the word "network of relations" in stead of "web of relations", which is also very suitable and coherent with our reading.

[25] This is partly a result of the influence from Hilbert's formalistic holism standpoint on the Foundations of Geometry (1999), from which Husserl learned that the geometrical primitives, such and point or line, are not defined separately but should be meaningful in the relation to the whole axiomatic system as the whole. See [Okada 2004]

[26] Husserl describes the term rewrite structure with standard as follows, for example.

> The numbers are the *standards of operation* [Operetionsetalons] in an defined operation-domain; these are the joints of a complete whole totality neither augmentable, nor diminishable, of unique and pairwisely different [untereinander nicht äquivalenten] operation-character, which are the lowest specific differences in this sphere of operation and which have the property that any real operation of its domain must have its provably [nachweislich] equivalent in an characterization of this whole. [*Hua XII*, p.475].

[27] On the Figure there are inequality edges between normal constructor terms. This corresponds to a proof of $\neg s = t$ in HusseOnrl's sense, which is in the convergent term rewrite sense; he emphasizes that "$\neg s = t$ is provable" is not in the logical provability sense, but in the sense of convergent rewriting. Namely, for two terms s and t $\neg s = t$ is provable in a (constructive) definite multiplicity when the normal terms of them are different (otherwise equal). Since the completeness in the sense of Husserl is based on the minimal term model, this means $\neg s = t$. We recall that he mainly the considers constructive multiplicity

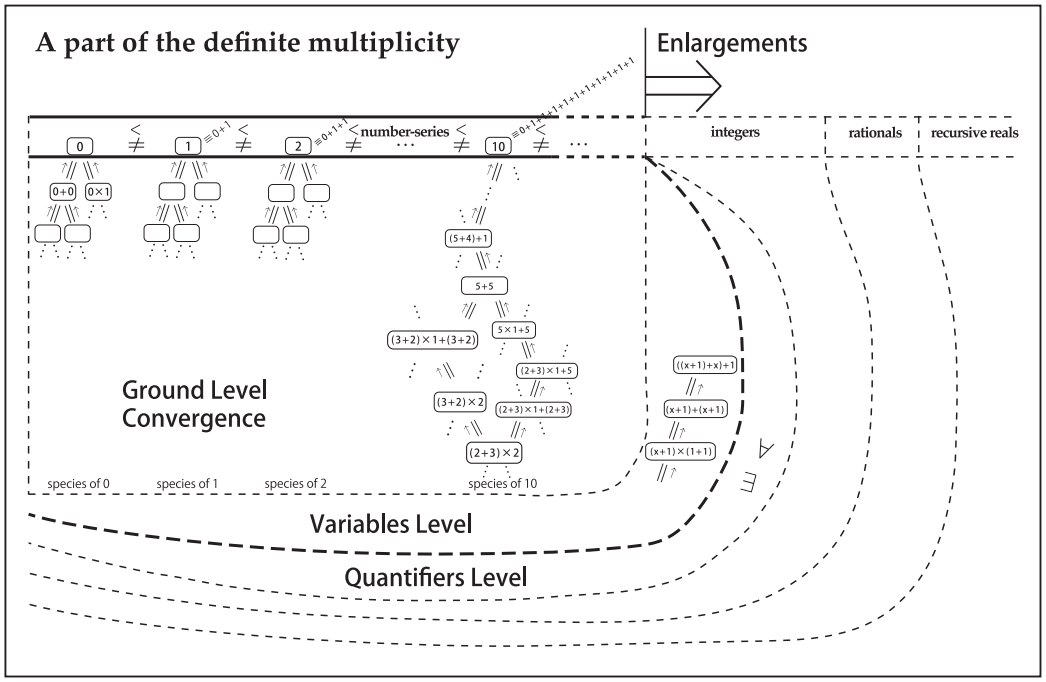


Figure 1 The above gives the reader a rough picture of Husserlian constructive definite multiplicity of the axiomatic system including axioms:

- $x + 0 = x$,
- $x + (y + 1) = (x + y) + 1$,
- $x \times 1 = x$,
- $x \times (y + 1) = (x \times y) + x$

Husserl believed that his completeness/definiteness works not only for the theories of natural numbers, of integers, of rationals, but also for the theories of real numbers and of complex numbers in the uniform way (although as we now know that it does not work with real number theory as the the decision of the primitive numerical equality-relation is not decidable anymore. Hence, it needs to be limited to the rewrite-computable or recursive reals (or in the case of an abstract real closed field system) if one wishes to defend his computational view of universal arithmetic.)

More About Completion Procedure

Husserl explains on the non-definite, namely non-convergent case that some suitable equational axioms should be added so that the multiplicity becomes convergent, in his sense that the

case as the ideal definite multiplicities. The different views about the linking axioms between Hilbert and Husserl becomes more clear when we understand. In the Hilbert sense, any term (composed by the linking axioms) exists, and the linking axioms are understood as existential axioms. This is because, $t = t$ implies $\exists x(t = x)$. On the other hand, for Husserl, term t composed by linking axiom only has an intentional meaning, or representation of concept of t. The existence of t is shown only when it reduced to a constructer term (namely, there is at least a rewriting path from t to a constructer term in the web structure of the constructive multiplicity web. See Section 2 above.

axiomatic system complete. He says that he addition should be done so that the "result" is the same, in other words, the equational provability is equivalent to the rewrite proof system, which means completion in the sense of Knuth-Bendix.

He explains the completion in a most detailed and reasonable way with respect to a constructive multiplicity, where the constructor-based normal forms are "pre-given" with the axiomatic setting (of Linking Axioms).

> [I]f any relation between them [two terms in a multiplicity] were ambiguous or undetermined [which means that the two terms go to different irreducible terms], I could then add the axioms which would introduce the determination; any undetermined relation should be, on the basis of the axioms, transformable into a determined relation. [*Hua XII*, p.497]

He tries to clarify this further and says:

> If there are two ways of determination which gives the same result [we read this that if there are two ways to reach different irreducible terms s and t from a term u], which shows both s and t are provably the same as u], we could then fix them arbitrarily [namely, one could add a rewrite rule from one irreducible term to the other.], then it should have an [additional] axiom which unites them. [*Hua XII*, p.498]

This is understood as a so called "Knuth-Bendix completion procedure" . The procedure was introduced in the 1960s from the computer scientific context by Knuth and Bendix [Knuth and Bendix 1970] and which was presented as a general procedure by Huet-Oppen in the 1980s Cf. [Dershowitz and Jouannaud 1990] for the general historical information and basic notions on the term rewrite theory in theoretical computational science, although Husserl's setting was the case of terminating (constructive) rewrite systems, [28]

Husserl's completion procedure is to add new equational axioms with a direction to make the underlying non-confluent term-rewriting system confluent, while the provability power of the axiomatic system unchanged; namely the additions of new axioms are redundant in the sense of logical provability, but necessary in the sense of computation.[29]

About the variables level

As mentioned in the previous Section, the equivalence between the rewrite-provability and the equational provability becomes delicate with the non-ground term rewrite case because one usually needs to add some additional non-rewrite deductive principle or inference rule in addition to the purely equational calculation axioms in order to deduce an algebraic equational proposition with variables . For example, $x + y = y + x$ is "true" in the sense of a (standard) model of arithmetic, but one usually needs the mathematical induction

[28] There were similar procedure defined in mathematics, especially in algebra-related fields independently, but the procedure appeared in mathematics had been very much dependent on a rather specific field, and even Knuth-Bendix' presentation was the word problem oriented. So, Huet-Oppen[Huet and Oppen 1980] seems a very first to propose it as a procedure on a term rewrite-oriented equational system in general. But, we would lime to point out the idea of the procedure appeared in Husserl (1901) for his theory of term rewriting even yet conceptually, in the context of philosophy of logic and mathematics.l

[29] When he explains the (completion) procedure in a most detail way Husserl presumes that the underlying axiomatic system has the constructors, hence a multiplicity has the constructor terms as irreducible normal forms. But a term in the multiplicity might property guarantees the uniqueness property of the irreducible terms, which is of a specific importance for Husserl as (the concept of) each irreducible term serves as a definite "specific difference" by the constructor based normal-irreducible terms

rule to prove it. Note that both $x = y$ and $y = x$ are irreducible terms in a simple (non-overlap) convergent rewrite system such as the primitive recursive rule system in the example above. the convergence does not correspond to the truth of the standard model on the standard/measure. Husserl were very much aware of the delicate issue of the variable level of definite multiplicity. Husserl confirms ground convergence of arithmetical systems mentioned in the previous Section. And he also tells convergence on the variable rewrite level of the system (without commutativity). But he also asked himself the precise reason why it is. In fact he asks himself in footnote 39 (at the very last footnote in the Double Lecture manuscript, of the Sschumann-Schumann edition) saying "Why?" which was added as the footnote for a passage where he explains the ground convergence of arithmetical system (of rationals). [Schuhmann and Schuhmann 2001].

We add a small remark about the further development of equational proof system of arithmetic. This line of systematic study of proof system for arithmetic began only more than 20 years later than Husserl by Skolem's Primitive Recursive Arithmetic[30]. But, Skolem's (rather informal) principle of mathematical induction required non-equational logical inference (implication or conditional) in addition to the purely equational language. (One could express it in terms of the natural-deduction style inference rule, Induction Rule, as below, with the non-local but global inference rule. See the Induction Rule in the footnote below.) It was a philosopher, Wittgenstein, who first reformulated the mathematical induction rule in an equational way, which is now called the Uniqueness Rule, without using logic (implication).[31] Wittgenstein read and studied Skolem carefully and proposed his equational Uniqueness Rule without logic, as an alternative representation of mathematical Induction in the 1920's. The hypothetical appearance of a proposition at the induction step is reduced to equational Uniqueness (Inference) Rule. It is by this Rule form of Induction that the algebraic rules, for example, $x + y = y + x$ with variables x and y, is equationally provable without logic, Wittgenstein's philosophy student, Goodstein, who was a constructivist mathematician, took the Wittgenstein's Uniqueness Rule as the basis of his Recursive Number Theory. He gave the equivalence proof between the Induction Rule and the Uniqueness Rule under the presence of logic (with implication-conditional) although Wittgenstein took it for granted in his philosophical discussions.[32] Goodstein also had a version of purely equational representation (Goodstein Induction) of the Induction Rule (see the footnote below). However, it requires additional axioms of positive minus $x \dot{-} x = 0$ (for natural numbers) and the absolute value function, which are , however,not direction oriented rewrite axioms (although of course equational). The spirit of this Goodstein's equational axiomatization of arithmetic was oriented by the line of Wittgenstein's formulation (i.e., reduction of logical implication into equational calculus). It was Lambek and some others much later who developed equational type systems with rewrite rules of Mal'cev operator (which plays the role of logical implication by rewrite rules). With See [Okada 1999] for the rewrite theoretic discussions of Goodstein and Lambek Uniqueness Rule, where it is exposed that one way of direction-oriented Lambek

[30] Also, in 1931, Gödel's incompleteness appeared and told that there are true universal proposition on the variables level which is not provable in any axiomatic consistent arithmetic. This means that even if it has a definite multiplicity in the strong sense, hence convergent on the variable level, there is always a "true" equality which cannot be reached by going down through in the definite multiplicity web.

[31] Note that Husserl and Wittgenstein are known as two of the most philosophers in the Western world in the 20th Century. It is interesting to see that both studied and researched deeply equational theories.

[32] Skolem's Primitive recursive Arithmetic was taken as the basis of their Finitist System by Hilbert-Bernays after the appearance of Gödel's Incompleteness Theorem. Then, Gödel extended it to higher types in his interpretation (Dialectica Interpretation) of Gentzen's consistency proof.

Uniqueness Rule makes the whole rewrite system confluent and the opposite way of orientation makes it terminating, on the variables levels of type theories. This fact tells that the type theories with uniqueness still have a semi-definite multiplicity (i.e., the computational content) in the sense of Husserl even on the variable-higher type levels, although they are never convergent, namely never truly definite, as the definiteness with the uniqueness rule conflicts with the incompleteness theorem of Gödel. [Okada 2004] [Marion and Okada 2012] for the philosophical discussion on the equational proofs on the variables level of Wittgenstein and Goodstein.[33]

Conclusion

We presented that the Husserl proposed, in 1901, his own view on the notion of completeness by modifying Hilbert's axiomatic system of arithmetic (1900). He gave a sufficient condition for solving the problem of justifying the use of imaginaries in mathematics, which can be understood as the conservation condition in the modern logical sense, The condition was given with two stages. On the first stage he gave the condition by the use of the notions of syntactic completeness and consistency. Then, on the second (higher) stage of his research and presentation, he explained the notion of completeness more precisely. He claimed that an axiomatic system is complete (in his sense) if and only if the multiplicity (manifold) of the system is "definite". The notion of multiplicity and that of definiteness of multiplicity are the key notions to understand the whole picture of Husserl's theory and his solution to the problem. In this paper, we clarified what is these key notions. We claimed that these notions are coherently understood by means of general term rewrite theory. In particular, a multiplicity is understood as relational-web (or tissue) where term rewrite proof-steps (moves) both of the closed terms level and general terms level are the basic part of the multiplicity-web. The definiteness corresponds to convergence of this part. He considered a constructors-based definite multiplicity an ideal definite multiplecity. This tells that Husserl's view of axiomatic (arithmeticcal) systems was very much oriented by the computational view and his view was very much advanced in terms of computation theory, which were developed much later in modern logic and theoretical computer science. Husserl also introduced completion procedure of the underlying rewrite system. He gave general conditions to enlarge a rewrite system with preserving conservation. We think that Husserl's notion of multiplicity and completeness successfully extract the rewrite based-computational content from a given

---

33

1. The induction rule :

$$(Induction)\qquad \frac{\begin{array}{cc}[f(x,y)=g(x,y)]\\ \vdots\\ f(x,0)=g(x,0) \quad f(x,Sy)=g(x,Sy)\end{array}}{f(x,y)=g(x,y)}$$

2. Uniqueness rule:

$$(Uniquness)\qquad \frac{f(x,0)=g(x,0) \quad f(x,Sy)=h(x,y,f(x,y)) \quad g(x,Sy)=h(x,y,g(x,y))}{f(x,y)=g(x,y)}$$

3. Goodstein Induction:

$$(Goodstein\ Ind.)\qquad \frac{f(x,0)=0 \quad (1\dot{-}fxy)\cdot fx(Sy)=0}{f(x,y)=0}$$

arithmetic axiom system. We characterized Hilbert's notion of completeness (of 1900) as the maximally expanded categorical model, while Husserl's notion of completeness as the minimal term model although Husserl's notion of multiplicity is not just a model but more like a type theoretic-proof theoretic structure (even limited to the first order terms), where proof formation steps and term formation steps are the basic parts of the multiplicity web. We also discussed potentials and limitation of Husserl's line of the research paradigm from the equational arithmetical point of view, in the domains of philosophy of mathematics and of theory of term rewriting.

Acknowledgments

This paper is dedicated to Professor Jaakko Hintikka. The author would like to express his sincere thanks to Mr. Yutaro Sugimoto for his kind efforts on the editorial assistance during the preparation of this paper. This version of the paper is prepared for an invited talk at the 24th International Conference on Rewriting Techniques and its Applications (RTA2013). The author would also like to express his sincere thanks to the RTA 2013 Program Committee, especially the chair, Professor Femke van Raamsdonk for giving the opportunity to the author. The author would also like to express his thanks to Professor Mirja Hartimo for her continuous encouragement to publish the author's work.

References

Cavaillès 1947 Cavaillès, Jean. 1947. *Sur la logique et les theories des sciences.* Paris: Presses universitaires de France.

daSilva 2000 daSilva, JairoJosé. 2000. Husserl's two notions of completeness. *Synthese* 125:417–438.

Dershowitz and Jouannaud 1990 Dershowitz, Nachum, and Jean-Pierre Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, VolumeB, 243–320. Elsevier.

Detlefsen 1947 Detlefsen, Michael 1986. *Hilbert's Program: An Essay on Mathematical Instrumentalism.* Springer.

Ewald 1996 Ewald, William 1996. *From Kant to Hilbert : a source book in the foundations of mathematics.* Oxford : Clarendon Press.

Gödel 1931 Gödel, Kurt. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik* 38:173–98.

Gödel 1947 Gödel, Kurt. 1947. What is Cantor's continuum problem? *The American Mathematical Monthly* 54:515–25.

Hartimo 2007 Hartimo, Mirja. 2007. Towards completeness: Husserl on theories of manifolds 1890–1901. *Synthese* 156:281–310.

Hartimo 2010 Hartimo, Mirja (ed.) 2010. Phenomenology and Mathematics. *Springer*

Hilbert 1900 Hilbert, David. 1900. Über den Zahlbegriff. *Jahresbericht der Deutschen Mathematiker-Vereinigung* 8:180–194.

Hilbert 1926 Hilbert, David. 1926. Über das Unendliche. *Mathematische Annalen* 95:161–190.

Hilbert and Bernays 1934 Hilbert, David, and Paul Bernays. 1934. *Grundlagen der Mathematik.* Volume1. Springer.

Hill 1995 Hill, ClaireOrtiz. 1995. Husserl and Hilbert on Completeness. In *From Dedekind to Gödel: Essays on the Development of the Foundations of Mathematics*, ed. J.Hintikka, 143–63. Dordrecht: Kluwer.

Huet and Oppen 1980 Huet, Gérard, and Derek C. Oppen. 1980. Equations and Rewrite Rules: A Survey. Research Report, STAN-CS-80-785, 56 pages. Stanford University.

Logische Untersuchungen 1 Husserl, Edmund. 1900. *Logische Untersuchungen. Erste Teil: Prolegomena zur reinen Logik*. Martinus Nijhoff. Husserliana Bd.XVIII.

Logische Untersuchungen 2 Husserl, Edmund. 1901. *Logische Untersuchungen. Zweite Teil: Untersuchungen zur Phänomenologie und Theorie der Erkenntnis*. Martinus Nijhoff. Husserliana Bd.XIX.

Ideen I Husserl, Edmund. 1913. *Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie. Erstes Buch: Allgemeine Einführung in die reine Phänomenologie (Ideen I)*. M. Niemeyer. Husserliana Bd.III. (1931. *Ideas: general introduction to pure phenomenology*. London: Allen & Unwin; New York: Humanities P. Translated by W.R. Boyce Gibson.)

FTL Husserl, Edmund. 1929. *Formale und Transzendentale Logik (FTL)*. M. Niemeyer. Husserliana Bd.XVII. (1969. *Formal and transcendental logic*. Martinus Nijhoff. Translated by Dorion Cairns.)

Hua XII Husserl, Edmund. 1970. *Husserliana XII (Hua XII)*. Den Haag: Martinus Nijhoff.

Knuth and Bendix 1970 Knuth, Donald E., and P. B. Bendix. 1970. Simple Word Problems in Universal Algebra. In *Computational Problems in Abstract Algebra (Proc. Conf., Oxford, 1967)*, 263–297. Pergamon Press.

Kreisel 1958 Kreisel, George. 1958. Mathematical significance of consistency proofs. *J. Symbolic Logic* 23: 155-182.

Majer 1997 Majer, Ulrich. 1997. Husserl and Hilbert on Completeness. *Synthese* 110:37–56.

Marion and Okada 2012 Marion, Mathieu. and Okada, Mitsuhiro. 2012. La philosophie des mathématiques de Wittgenstein. In *Lectures de Wittgenstein*, eds Chauviré Christiane et Plaud Sabine 79–104. Ellipses

Okada 1987 Okada, Mitsuhiro. 1987. Husserl's solution to the 'Philosophical-Mathematical final theme' and its relation with the Göttingen School of logical philosophy. *Journal of Philosophical Society of Japan* 37:210–221.

Okada 1999 Okada, Mitsuhiro, and Philip J. Scott. 1999. A Note on Rewriting Theory for Uniqueness of Iteration. *Theory and Applications of Categories* 6(4):47–64.

Okada 2004 Okada, Mitsuhiro. 2004. Correspondence-theory and Contextual-theory of Meaning in Husserl and Wittgenstein. In *Husserl et Wittgenstein*, ed. J.Benoist and S.Laugier, 27–69. Paris: Olms.

Okada 2007 Okada, Mitsuhiro. 2007. On Wittgenstein's Remarks on Recursive Proofs: A Preliminary Report. In *Essays in the Foundations of Logical and Phenomenological Studies, Interdisciplinary Series on Reasoning Studies, Vol. 3*, 121–131. Tokyo:Keio University Press.

Schuhmann and Schuhmann 2001 Schuhmann, Elisabeth, and Karl Schuhmann. 2001. Husserls Manuskripte zu seinem Göttinger Doppelvortrag von 1901. *Husserl Studies* 17:87–123.

Skolem 1923 Skolem, Thoralf. 1923. Begründung der elementären Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnugsbereich. *Videnskapsselskapets skrifter, I. Matematisk-naturvidenskabelig klasse*, no. 6.

PG Wittgenstein, Ludwig. 1974. *Philosophical Grammar (PG)*. R. Rhees (ed.), A. Kenny (trans.) Oxford: Blackwell.

PR Wittgenstein, Ludwig. 1975. *Philosophical Remarks (PR)*. R. Rhees (ed.), R. Hargreaves and R. White (trans.) Oxford: Blackwell.

Evidence Normalization in System FC (Invited Talk)

Dimitrios Vytiniotis and Simon Peyton Jones

Microsoft Research, Cambridge
dimitris@microsoft.com, simonpj@microsoft.com

Abstract

System FC is an explicitly typed language that serves as the target language for Haskell source programs. System FC is based on System F with the addition of erasable but explicit type equality proof witnesses. Equality proof witnesses are generated from type inference performed on source Haskell programs. Such witnesses may be very large objects, which causes performance degradation in later stages of compilation, and makes it hard to debug the results of type inference and subsequent program transformations. In this paper we present an equality proof simplification algorithm, implemented in GHC, which greatly reduces the size of the target System FC programs.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, F.3.3 Studies of Program Constructs

Keywords and phrases Haskell, type functions, system FC

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.20

1 Introduction

A statically-typed intermediate language brings a lot of benefits to a compiler: it is free from the design trade-offs that come with source language features; types can inform optimisations; and type checking programs in the intermediate language provides a powerful consistency check on each stage of the compiler.

The Glasgow Haskell Compiler (GHC) has just such an intermediate language, which has evolved from System F to System FC [16, 20] to accommodate the source-language features of *GADTs* [6, 15, 13] and *type families* [9, 3]. The key feature that allows System FC to accomodate GADTs and type families is its use of explicit *coercions* that witness the equality of two syntactically-different types. Coercions are erased before runtime but, like types, serve as a static consistency proof that the program will not "go wrong".

In GHC, coercions are produced by a fairly complex type inference (and proof inference) algorithm that elaborates source Haskell programs into FC programs [19]. Furthermore, coercions undergo major transformations during subsequent program optimization passes. As a consequence, they can become very large, making the compiler bog down. This paper describes how we fixed the problem:

- Our main contribution is a novel coercion simplification algorithm, expressed as a rewrite system, that allows the compiler to replace a coercion with an equivalent but much smaller one (Section 4).
- Coercion simplification is important in practice. We encountered programs whose unsimplified coercion terms grow to many times the size of the actual executable terms, to the point where GHC choked and ran out of heap. When the simplifier is enabled, coercions simplify to a small fraction of their size (Section 5).

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 20–38

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


c	$\in$	Coercion variables	
x	$\in$	Term variables	
e, u	$::=$	$x \mid l \mid \lambda x{:}\sigma\,.\,e \mid e\ u$	
	$\mid$	$\Lambda a{:}\eta\,.\,e \mid e\ \phi$	Type polymorphism
	$\mid$	$\lambda c{:}\tau\,.\,e \mid e\ \gamma$	Coercion abstraction/application
	$\mid$	$K \mid \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	Constructors and case expressions
	$\mid$	$\mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ u$	Let binding
	$\mid$	$e \triangleright \gamma$	Cast
p	$::=$	$K\ \overline{c{:}\tau}\ \overline{x{:}\tau}$	Patterns

Figure 1 Syntax of System FC (Terms).

- To get these benefits, coercion simplification must take user-declared equality axioms into account, but the simplifier *must never loop* while optimizing a coercion – no matter which axioms are declared by users. Proof normalization theorems are notoriously hard, but we present such a theorem for our coercion simplification. (Section 6)

Equality proof normalization was first studied in the context of monoidal categories and we give pointers to early work in Section 7 – this work in addition addresses the simplification of open coercions containing variables and arbitrary user-declared axioms.

2 An overview of System FC

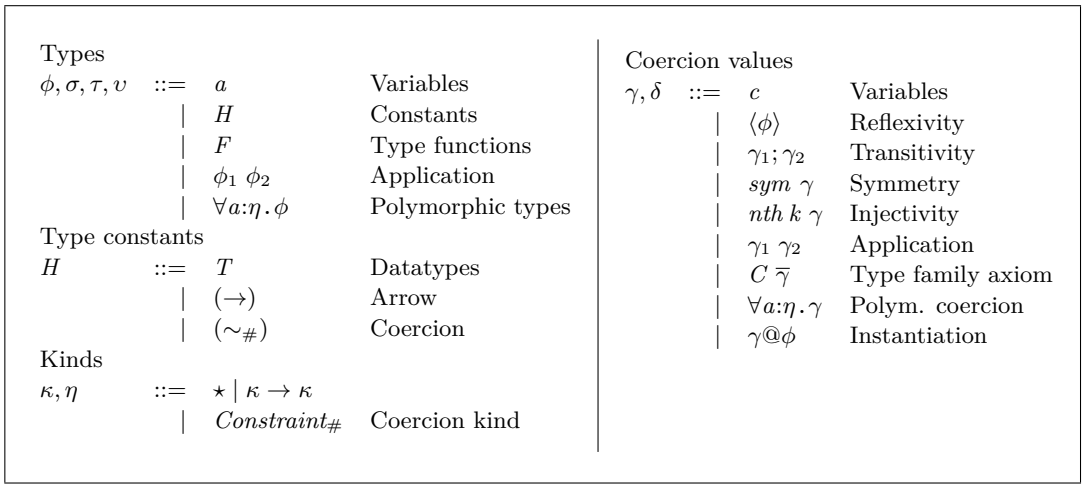
We begin by reviewing the role of an intermediate language. GHC desugars a rich, complex source language (Haskell) into a small, simple intermediate language. The source language, Haskell, is *implicitly typed*, and a type inference engine figures out the type of every binder and sub-expression. To make type inference feasible, Haskell embodies many somewhat ad-hoc design compromises; for example, λ-bound variables are assigned monomorphic types. By contrast, the intermediate language is simple, uniform, and *explicitly typed*. It can be typechecked by a simple, linear time algorithm. The type inference engine *elaborates* the implicitly-typed Haskell program into an explicitly-typed FC program.

To make this concrete, Figure 1 gives the syntax of System FC, the calculus implemented by GHC's intermediate language. The term language is mostly conventional, consisting of System F, together with let bindings, data constructors and case expressions. The syntax of a term encodes its typing derivation: every binder carries its type, and type abstractions $\Lambda a{:}\eta\,.\,e$ and type applications $e\,\phi$ are explicit.

The types and kinds of the language are given in Figure 2. Types include variables (a) and constants H (such as `Int` and `Maybe`), type applications (such as `Maybe Int`), and polymorphic types ($\forall a{:}\eta\,.\,\phi$). The syntax of types also includes *type functions* (or *type families* in the Haskell jargon), which are used to express type level computation. For instance the following declaration in source Haskell:

```
type family F (a :: *) :: a
type instance F [a] = a
```

introduces a type function F at the level of System FC. The accompanying `instance` line asserts that any expression of type `F [a]` can be viewed as having type `a`. We shall see in Section 2.2 how this fact is expressed in FC. Finally type constants include datatype constructors (T) but also arrow ($\rightarrow$) as well as a special type constructor $\sim_{\#}$ whose role we



Types

$\phi, \sigma, \tau, \upsilon$	::=	a	Variables
	|	H	Constants
	|	F	Type functions
	|	$\phi_1\ \phi_2$	Application
	|	$\forall a{:}\eta\,.\,\phi$	Polymorphic types

Type constants

H	::=	T	Datatypes
	|	$(\rightarrow)$	Arrow
	|	$(\sim_\#)$	Coercion

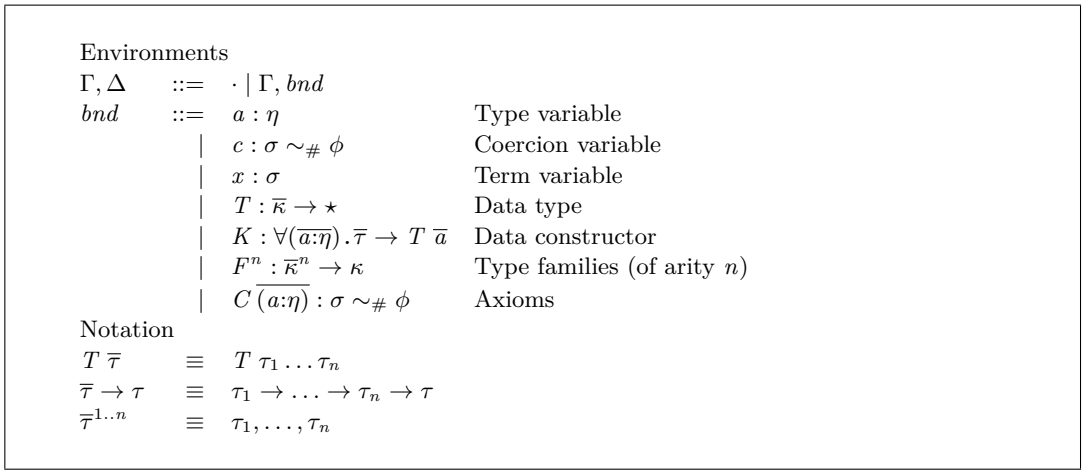
Kinds

κ, η	::=	$\star \mid \kappa \rightarrow \kappa$	
	|	$\mathit{Constraint}_\#$	Coercion kind

Coercion values

γ, δ	::=	c	Variables
	|	$\langle\phi\rangle$	Reflexivity
	|	$\gamma_1;\gamma_2$	Transitivity
	|	$\mathit{sym}\ \gamma$	Symmetry
	|	$\mathit{nth}\ k\ \gamma$	Injectivity
	|	$\gamma_1\ \gamma_2$	Application
	|	$C\ \overline{\gamma}$	Type family axiom
	|	$\forall a{:}\eta\,.\,\gamma$	Polym. coercion
	|	$\gamma@\phi$	Instantiation

Figure 2 Syntax of System FC (types and coercions).



Environments

Γ, Δ	::=	$\cdot \mid \Gamma, \mathit{bnd}$	
bnd	::=	$a : \eta$	Type variable
	|	$c : \sigma \sim_\# \phi$	Coercion variable
	|	$x : \sigma$	Term variable
	|	$T : \overline{\kappa} \rightarrow \star$	Data type
	|	$K : \forall(\overline{a{:}\eta})\,.\,\overline{\tau} \rightarrow T\ \overline{a}$	Data constructor
	|	$F^n : \overline{\kappa}^n \rightarrow \kappa$	Type families (of arity n)
	|	$C\ \overline{(a{:}\eta)} : \sigma \sim_\# \phi$	Axioms

Notation

$T\ \overline{\tau}$	$\equiv$	$T\ \tau_1 \ldots \tau_n$
$\overline{\tau} \rightarrow \tau$	$\equiv$	$\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$
$\overline{\tau}^{1..n}$	$\equiv$	$\tau_1, \ldots, \tau_n$

Figure 3 Syntax of System FC (Auxiliary definitions).

explain in the following section. The kind language includes the familiar $\star$ and $\kappa_1 \rightarrow \kappa_2$ kinds but also a special kind called $\mathit{Constraint}_\#$ that we explain along with the $\sim_\#$ constructor.

The typing rules for System FC are given in Figure 4. We urge the reader to consult [16, 20] for more examples and intuition.

2.1 Coercions

The unusual feature of FC is the use of coercions. The term $e \triangleright \gamma$ is a cast, that converts a term e of type τ to one of type ϕ (rule ECast in Figure 4). The coercion γ is a *witness*, or *proof*, providing evidence that τ and ϕ are equal types – that is, γ has type $\tau \sim_\# \phi$. We use the symbol "$\sim_\#$" to denote type equality[1]. The syntax of coercions γ is given in Figure 2, and their typing rules in Figure 6. For uniformity we treat $\sim_\#$ as an ordinary type constructor, with kind $\kappa \rightarrow \kappa \rightarrow \mathit{Constraint}_\#$ (Figure 5).

To see casts in action, consider this Haskell program which uses GADTs:

[1] The "#" subscript is irrelevant for this paper; the interested reader may consult [18] to understand the related type equality $\sim$, and the relationship between $\sim$ and $\sim_\#$.

$$\boxed{\Gamma \vdash^{\mathsf{tm}} e : \tau}$$

$$\frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash^{\mathsf{tm}} x : \tau}\ \textsc{EVar} \qquad \frac{(K{:}\sigma) \in \Gamma}{\Gamma \vdash^{\mathsf{tm}} K : \sigma}\ \textsc{ECon}$$

$$\frac{\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{ty}} \sigma : \star}{\Gamma \vdash^{\mathsf{tm}} \lambda x{:}\sigma\,.\,e : \sigma \to \tau}\ \textsc{EAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \sigma \to \tau \quad \Gamma \vdash^{\mathsf{tm}} u : \sigma}{\Gamma \vdash^{\mathsf{tm}} e\ u : \tau}\ \textsc{EApp}$$

$$\frac{\Gamma, (c{:}\sigma) \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{ty}} \sigma : \mathit{Constraint}_{\#}}{\Gamma \vdash^{\mathsf{tm}} \lambda c{:}\sigma\,.\,e : \sigma \to \tau}\ \textsc{ECAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : (\sigma_1 \sim_{\#} \sigma_2) \to \tau \quad \Gamma \vdash^{\mathsf{co}} \gamma : \sigma_1 \sim_{\#} \sigma_2}{\Gamma \vdash^{\mathsf{tm}} e\ \gamma : \tau}\ \textsc{ECApp}$$

$$\frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \Lambda a{:}\eta\,.\,e : \forall a{:}\eta\,.\,\tau}\ \textsc{ETAbs} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \forall a{:}\eta\,.\,\tau \quad \Gamma \vdash^{\mathsf{ty}} \phi : \eta}{\Gamma \vdash^{\mathsf{tm}} e\ \phi : \tau[\phi/a]}\ \textsc{ETApp}$$

$$\frac{\Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} u : \sigma \quad \Gamma, (x{:}\sigma) \vdash^{\mathsf{tm}} e : \tau}{\Gamma \vdash^{\mathsf{tm}} \mathbf{let}\ x{:}\sigma = u\ \mathbf{in}\ e : \tau}\ \textsc{ELet} \qquad \frac{\Gamma \vdash^{\mathsf{tm}} e : \tau \quad \Gamma \vdash^{\mathsf{co}} \gamma : \tau \sim_{\#} \phi}{\Gamma \vdash^{\mathsf{tm}} e \rhd \gamma : \phi}\ \textsc{ECast}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{tm}} e : T\ \overline{\kappa}\ \overline{\sigma} \\ \text{For each branch } K\ \overline{x{:}\tau} \to u \\ \quad (K{:}\forall(\overline{a{:}\eta_a})\,.\,\overline{\sigma_1 \sim_{\#} \sigma_2} \to \overline{\tau} \to T\ \overline{a}) \in \Gamma \\ \quad \phi_i = \tau_i[\overline{\sigma}/\overline{a}] \\ \quad \phi_{1i} = \sigma_{1i}[\overline{\sigma}/\overline{a}] \\ \quad \phi_{2i} = \sigma_{2i}[\overline{\sigma}/\overline{a}] \quad \Gamma, \overline{c{:}\phi_1 \sim_{\#} \phi_2}\ \overline{x{:}\phi} \vdash^{\mathsf{tm}} u : \sigma\end{array}}{\Gamma \vdash^{\mathsf{tm}} \mathbf{case}\ e\ \mathbf{of}\ \overline{K\ (\overline{c{:}\sigma_1 \sim_{\#} \sigma_2})\ (\overline{x{:}\tau}) \to u} : \sigma}\ \textsc{ECase}$$

Figure 4 Well-formed terms.

$$\boxed{\Gamma \vdash^{\mathsf{ty}} \tau : \kappa}$$

$$\frac{(a{:}\eta) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} a : \eta}\ \textsc{TVar} \qquad \frac{(T{:}\kappa) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} T : \kappa}\ \textsc{TData} \qquad \frac{(F{:}\kappa) \in \Gamma}{\Gamma \vdash^{\mathsf{ty}} F : \kappa}\ \textsc{TFun}$$

$$\frac{\kappa_1, \kappa_2 \in \{\mathit{Constraint}_{\#}, \star\}}{\Gamma \vdash^{\mathsf{ty}} (\to) : \kappa_1 \to \kappa_2 \to \star}\ \textsc{TArr} \qquad \frac{}{\Gamma \vdash^{\mathsf{ty}} (\sim_{\#}) : \kappa \to \kappa \to \mathit{Constraint}_{\#}}\ \textsc{TEqPred}$$

$$\frac{\Gamma \vdash^{\mathsf{ty}} \phi_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash^{\mathsf{ty}} \phi_2 : \kappa_1}{\Gamma \vdash^{\mathsf{ty}} \phi_1\ \phi_2 : \kappa_2}\ \textsc{TApp} \qquad \frac{\Gamma, (a{:}\eta) \vdash^{\mathsf{ty}} \tau : \star}{\Gamma \vdash^{\mathsf{ty}} \forall a{:}\eta\,.\,\tau : \star}\ \textsc{TAll}$$

Figure 5 Well-formed types.

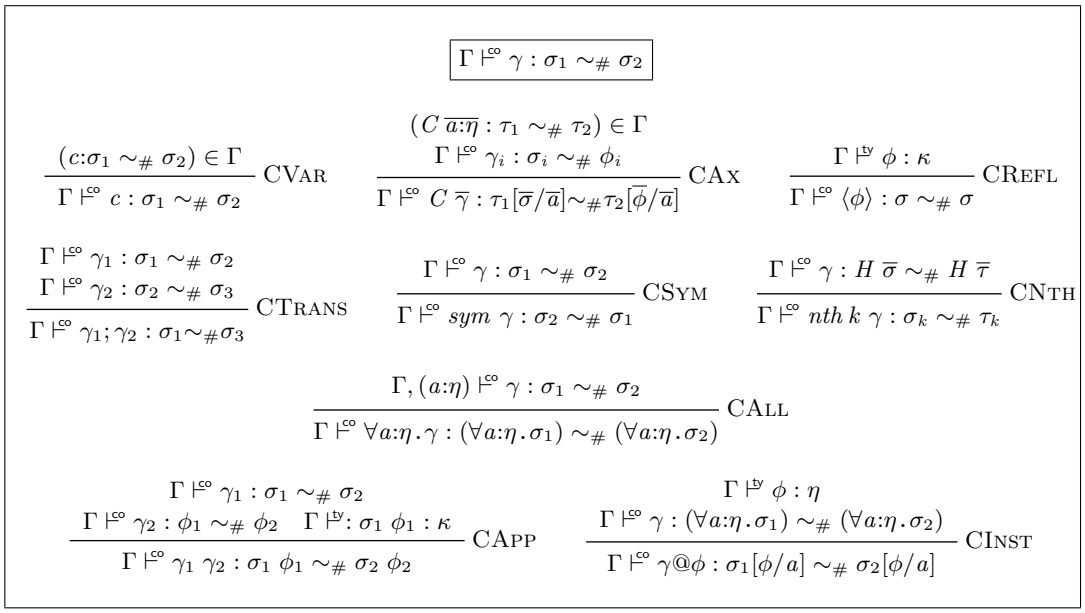


Figure 6 Well-formed coercions.

```
data T a where                f :: T a -> [a]
   T1 :: Int -> T Int         f (T1 x) = [x+1]
   T2 :: a -> T a             f (T2 v) = [v]

                              main = f (T1 4)
```

We regard the GADT data constructor `T1` as having the type

$$\texttt{T1} : \forall a.(a \sim_\# \texttt{Int}) \to \texttt{Int} \to \texttt{T}\ a$$

So in FC, `T1` takes three arguments: a type argument to instantiate a, a coercion witnessing the equivalence of a and `Int`, and a value of type `Int`. Here is the FC elaboration of `main`:

```
main = f Int (T1 Int <Int> 4)
```

The coercion argument has kind (`Int` $\sim_\#$ `Int`), for which the evidence is just $\langle$`Int`$\rangle$ (reflexivity). Similarly, pattern-matching on `T1` binds two variables: a coercion variable, and a term variable. Here is the FC elaboration of function `f`:

```
f = /\(a:*). \(x:T a).
    case x of
      T1 (c:a ~# Int) (n:Int) -> (Cons (n+1) Nil) |> sym [c]
      T2 (v:a)                -> Cons v Nil
```

The cast converts the type of the result from `[Int]` to `[a]`. The coercion *sym* $[c]$ is evidence for (or a proof of) the equality of these types, using coercion `c`, of type (`a` $\sim_\#$ `Int`).

2.2 Typing coercions

Figure 6 gives the typing rules for coercions. The rules include unsurprising cases for reflexivity (CRefl), symmetry (CSym), and transitivity (CTrans). Rules CAll and CApp allow us to construct coercions on more complex types from coercions on simpler types.

Rule CInst instantiates a coercion between two ∀-types, to get a coercion between two instantiated types. Rule CVar allows us to use a coercion that has been introduced to the context by a coercion abstraction ($\lambda c{:}\tau{\sim_\#}\phi\,.\,e$), or a pattern match against a GADT (as in the example above).

Rule CAx refers to instantiations of *axioms*. In GHC, axioms can arise as a result of *newtype* or *type family* declarations. Consider the following code:

```
newtype N a = MkN (a -> Int)

type family F (x :: *) :: *
type instance F [a]  = a
type instance F Bool = Char
```

N is a *newtype* (part of the original Haskell 98 definition), and is desugared to the following FC coercion axiom:

$$C_N\ a \quad : \quad N\ a \sim_\# a \to \texttt{Int}$$

which provides evidence of the equality of types $(N\ a)$ and $(a \to \texttt{Int})$.

In the above Haskell code, F is a *type family* [4, 3], and the two `type instance` declarations above introduce two FC coercion axioms:

$$\begin{array}{rcl} C_1\ a & : & F\ [a] \sim_\# a \\ C_2 & : & F\ \texttt{Bool} \sim_\# \texttt{Char} \end{array}$$

Rule CAx describes how these axioms may be used to create coercions. In this particular example, if we have $\gamma : \tau \sim_\# \sigma$, then we can prove that $C_1\ \gamma : F\ [\tau] \sim_\# \sigma$. Using such coercions we can get, for example, that $(3 \triangleright \mathit{sym}\,(C_1\ \langle\texttt{Int}\rangle)) : F\ [\texttt{Int}]$.

Axioms always appear saturated in System FC, hence the syntax $C\,\overline{\gamma}$ in Figure 2.

3 The problem with large coercions

System FC terms arise as the result of elaboration of source language terms, through type inference. Type inference typically relies on a *constraint solver* [19] which produces System FC witnesses of equality (coercions), that in turn decorate the elaborated term. The constraint solver is not typically concerned with producing small or readable witnesses; indeed GHC's constraint solver can produce large and complex coercions. These complex coercions can make the elaborated term practically impossible to understand and debug.

Moreover, GHC's optimiser transforms well-typed FC terms. Insofar as these transformations involve coercions, the coercions *themselves* may need to be transformed. If you think of the coercions as little proofs that fragments of the program are well-typed, then the optimiser must maintain the proofs as it transforms the terms.

3.1 How big coercions arise

The trouble is that *term-level optimisation tends to make coercions bigger*. The full details of these transformations are given in the so called *push* rules in our previous work [20], but we illustrate them here with an example. Consider this term:

$$(\lambda x.e \triangleright \gamma)\ a$$

where

$$\begin{array}{rcl} \gamma & : & (\sigma_1 \to \tau_1) \sim_\# (\sigma_2 \to \tau_2) \\ a & : & \sigma_2 \end{array}$$

We would like to perform the beta reduction, but the cast is getting in the way. No matter! We can transform thus:

$$\begin{array}{rl} & (\lambda x.e \rhd \gamma)\, a \\ = & ((\lambda x.e)\, (a \rhd sym\, (nth\, 0\, \gamma))) \rhd nth\, 1\, \gamma \end{array}$$

From the coercion γ we have derived two coercions whose syntactic form is larger, but whose types are smaller:

$$\begin{array}{rcl} \gamma & : & (\sigma_1 \to \tau_1) \sim_\# (\sigma_2 \to \tau_2) \\ sym\, (nth\, 0\, \gamma) & : & \sigma_2 \sim_\# \sigma_1 \\ nth\, 1\, \gamma & : & \tau_1 \sim_\# \tau_2 \end{array}$$

Here we make use of the coercion combinators *sym*, which reverses the sense of the proof; and $nth\, i$, which from a proof of $T\, \overline{\sigma} \sim_\# T\, \overline{\tau}$ gives a proof of $\sigma_i \sim_\# \tau_i$. Finally, we use the derived coercions to cast the argument and result of the function separately. Now the lambda is applied directly to an argument (without a cast in the way), so β-reduction can proceed as desired. Since β-reduction is absolutely crucial to the optimiser, this ability to "push coercions out of the way" is fundamental. Without it, the optimiser is hopelessly compromised.

A similar situation arises with `case` expressions:

$$\texttt{case}\, (K\ e_1 \rhd \gamma)\, \texttt{of}\, \{\ldots;\ K\ x \to e_2; \ldots\}$$

where K is a data constructor. Here we want to simplify the `case` expression, by picking the correct alternative $K\, x \to e_2$, and substituting e_1 for x. Again the coercion gets in the way, but again it is possible to push the coercion out of way.

3.2 How coercions can be simplified

Our plan is to simplify complicated coercion terms into simpler ones, using rewriting. Here are some obvious rewrites we might think of immediately:

$$\begin{array}{rcll} sym\, (sym\ \gamma) & \rightsquigarrow & \gamma & \\ \gamma; sym\ \gamma & \rightsquigarrow & \langle \tau \rangle & \text{if}\, \gamma : \tau \sim_\# \phi \end{array}$$

But ther are much more complicated rewrites to consider. Consider these coercions, where C_N is the axiom generated by the newtype coercion in Section 2.2:

$$\begin{array}{rcl} \gamma_1 & : & \tau_1 \sim_\# \tau_2 \\ \gamma_2 = sym\, (C_N\, \langle \tau_1 \rangle) & : & (\tau_1 \to \texttt{Int}) \sim_\# (N\, \tau_1) \\ \gamma_3 = N\, \langle \gamma_1 \rangle & : & (N\, \tau_1) \sim_\# (N\, \tau_2) \\ \gamma_4 = C_N\, \langle \tau_2 \rangle & : & (N\, \tau_2) \sim_\# (\tau_2 \to \texttt{Int}) \\ \\ \gamma_5 = \gamma_2; \gamma_3; \gamma_4 & : & (\tau_1 \to \texttt{Int}) \sim_\# (\tau_2 \to \texttt{Int}) \end{array}$$

Here γ_2 takes a function, and wraps it in the newtype; then γ_3 coerces that newtype from $N\, \tau_1$ to $N\, \tau_2$; and γ_4 unwraps the newtype. Composing the three gives a rather large, complicated coercion $\gamma_2; \gamma_3; \gamma_4$. *But its type is pretty simple*, and indeed the coercion $\gamma_1 \to \langle \texttt{Int} \rangle$ is a much simpler witness of the same equality. The rewrite system we present shortly will rewrite the former to the latter.

Finally, here is an actual example taken from a real program compiled by GHC (don't look at the details!):

$$\begin{array}{ll} & \texttt{Mut}\,\langle v\rangle\,(sym\,(C_{StateT}\,\langle s\rangle))\,\langle a\rangle \\ & ;\,sym\,(nth\,0\,((\forall wtb.\,\texttt{Mut}\,\langle w\rangle\,(sym\,(C_{StateT}\,\langle t\rangle))\,\langle b\rangle \to \langle \texttt{ST}\,t\,(w\;b)\rangle)@v@s@a) \\ \rightsquigarrow & \langle \texttt{Mut}\;v\;s\;a\rangle \end{array}$$

As you can see, the shrinkage in coercion size can be dramatic.

4 Coercion simplification

We now proceed to the details of our coercion simplification algorithm. We note that the design of the algorithm is guided by empirical evidence of its effectiveness on actual programs and that other choices might be possible. Nevertheless, we formally study the properties of this algorithm, namely we will show that it preserves validity of coercions and terminates – even when the rewrite system induced by the axioms is not strongly normalizing.

4.1 Simplification rules

Coercion simplification is given as a non-deterministic relation in Figure 7 and Figure 8 In these two figures we use some syntactic conventions: Namely, for sequences of coercions $\overline{\gamma}_1$ and $\overline{\gamma}_2$, we write $\overline{\gamma_1;\gamma_2}$ for the sequence of pointwise transitive compositions and $sym\;\overline{\gamma}_1$ for pointwise application of symmetry. We write $nontriv(\gamma)$ iff γ *contains* some variable c or axiom application $C\;\overline{\gamma}$.

We define coercion evaluation contexts, $\mathcal{G}$, as coercion terms with holes inside them. The syntax of $\mathcal{G}$ allows us to rewrite anywhere inside a coercion. The main coercion evaluation rule is CoEval. If we are given a coercion γ, we first decompose it to some evaluation context $\mathcal{G}$ with γ_1 in its hole. Rule CoEval works up to associativity of transitive composition; for example, we will allow the term $(\gamma_1;\gamma_2;);\gamma_3$ to be written as $\mathcal{G}[\gamma_2;\gamma_3]$ where $\mathcal{G} = \gamma_1;\Box$. This treatment of transitivity is extremely convenient, but we must be careful to ensure that our argument for termination remains robust under associativity (Section 6). Once we have figured out a decomposition $\mathcal{G}[\gamma_1]$, CoEval performs a single step of rewriting $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ and simply return $\mathcal{G}[\gamma_2]$. Since we are allowed to rewrite coercions under a type environment ($\forall a{:}\eta\,.\,\mathcal{G}$ is a valid coercion evaluation context), Δ (somewhat informally) enumerates the type variables bound by $\mathcal{G}$. For instance we should be allowed to rewrite $\forall a{:}\eta\,.\,\gamma_1$ to $\forall a{:}\eta\,.\,\gamma_2$. This can happen if $(a{:}\eta)| - \gamma_1 \rightsquigarrow \gamma_2$. The precondition $\Delta \vDash^{\text{co}} \gamma_1 : \sigma \sim_{\#} \phi$ of rule CoEval ensures that this context corresponds to the decomposition of γ into a context and γ_1. Moreover, the Δ is passed on to the $\rightsquigarrow$ relation, since some of the rules of the $\rightsquigarrow$ relation that we will present later may have to consult the context Δ to establish preconditions for rewriting.

The soundness property for the $\longrightarrow$ relation is given by the following theorem.

▶ **Theorem 1** (Coercion subject reduction). *If* $\vDash^{\text{co}} \gamma_1 : \sigma \sim_{\#} \phi$ *and* $\gamma_1 \longrightarrow \gamma_2$ *then* $\vDash^{\text{co}} \gamma_2 : \sigma \sim_{\#} \phi$.

The rewriting judgement $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ satisfies a similar property.

▶ **Lemma 2.** *If* $\Delta \vDash^{\text{co}} \gamma_1 : \sigma \sim_{\#} \phi$ *and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *then* $\Delta \vDash^{\text{co}} \gamma_2 : \sigma \sim_{\#} \phi$.

To explain coercion simplification, we now present the reaction rules for the $\rightsquigarrow$ relation, organized in several groups.

Coercion evaluation contexts $\mathcal{G} ::= \Box \mid \mathcal{G}\ \gamma \mid \gamma\ \mathcal{G} \mid C\ \overline{\gamma}_1 \mathcal{G} \overline{\gamma}_2 \mid sym\ \mathcal{G} \mid \forall a{:}\eta.\mathcal{G} \mid \mathcal{G}@\tau \mid \mathcal{G};\gamma \mid \gamma;\mathcal{G}$

$$\frac{\gamma \cong \mathcal{G}[\gamma_1] \text{ modulo associativity of } (;) \quad \Delta \vdash^{\text{co}} \gamma_1 : \sigma \sim_{\#} \phi \quad \Delta \vdash \gamma_1 \rightsquigarrow \gamma_2}{\gamma \longrightarrow \mathcal{G}[\gamma_2]}\ \textsc{CoEval}$$

$\boxed{\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2}$

Reflexivity rules

ReflApp	$\Delta \vdash \langle\phi_1\rangle\ \langle\phi_2\rangle$	$\rightsquigarrow$	$\langle\phi_1\ \phi_2\rangle$
ReflAll	$\Delta \vdash \forall a{:}\eta.\langle\phi\rangle$	$\rightsquigarrow$	$\langle\forall a{:}\eta.\phi\rangle$
ReflElimL	$\Delta \vdash \langle\phi\rangle;\gamma$	$\rightsquigarrow$	γ
ReflElimR	$\Delta \vdash \gamma;\langle\phi\rangle$	$\rightsquigarrow$	γ

Eta rules

EtaAllL	$\Delta \vdash ((\forall a{:}\eta.\gamma_1);\gamma_2)@\phi$	$\rightsquigarrow$	$\gamma_1[\phi/a];(\gamma_2@\phi)$
EtaAllR	$\Delta \vdash (\gamma_1;(\forall a{:}\eta.\gamma_2))@\phi$	$\rightsquigarrow$	$\gamma_1@\phi;\gamma_2[\phi/a]$
EtaNthL	$\Delta \vdash nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma};\gamma)$	$\rightsquigarrow$	$\begin{cases} nth\ k\ \gamma & \text{if } k \le \ell \\ \gamma_{k-\ell};nth\ k\ \gamma & \text{otherwise} \end{cases}$
EtaNthR	$\Delta \vdash nth\ k\ (\gamma;\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma})$	$\rightsquigarrow$	$\begin{cases} nth\ k\ \gamma & \text{if } k \le \ell \\ nth\ k\ \gamma;\gamma_{k-\ell} & \text{otherwise} \end{cases}$

Symmetry rules

SymRefl	$\Delta \vdash sym\ \langle\phi\rangle$	$\rightsquigarrow$	$\langle\phi\rangle$
SymAll	$\Delta \vdash sym\ (\forall a{:}\eta.\gamma)$	$\rightsquigarrow$	$\forall a{:}\eta.\ sym\ \gamma$
SymApp	$\Delta \vdash sym\ (\gamma_1\ \gamma_2)$	$\rightsquigarrow$	$(sym\ \gamma_1)\ (sym\ \gamma_2)$
SymTrans	$\Delta \vdash sym\ (\gamma_1;\gamma_2)$	$\rightsquigarrow$	$(sym\ \gamma_2);(sym\ \gamma_1)$
SymSym	$\Delta \vdash sym\ (sym\ \gamma)$	$\rightsquigarrow$	γ

Reduction rules

RedNth	$\Delta \vdash nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma})$	$\rightsquigarrow$	$\begin{cases} \langle\tau_k\rangle & \text{if } k \le \ell \\ \gamma_{k-\ell} & \text{otherwise} \end{cases}$
RedInstCo	$\Delta \vdash (\forall a{:}\eta.\gamma)@\phi$	$\rightsquigarrow$	$\gamma[\phi/a]$
RedInstTy	$\Delta \vdash \langle\forall a{:}\eta.\tau\rangle@\phi$	$\rightsquigarrow$	$\langle\tau[\phi/a]\rangle$

Push transitivity rules

PushApp	$\Delta \vdash (\gamma_1\ \gamma_2);(\gamma_3\ \gamma_4)$	$\rightsquigarrow$	$(\gamma_1;\gamma_3)\ (\gamma_2;\gamma_4)$	
PushAll	$\Delta \vdash (\forall a{:}\eta.\gamma_1);(\forall a{:}\eta.\gamma_2)$	$\rightsquigarrow$	$\forall a{:}\eta.\gamma_1;\gamma_2$	
PushInst	$\Delta \vdash (\gamma_1@\tau);(\gamma_2@\tau)$	$\rightsquigarrow$	$(\gamma_1;\gamma_2)@\tau$	when $\Delta \vdash^{\text{co}} \gamma_1;\gamma_2 : \sigma_1 \sim_{\#} \sigma_2$
PushNth	$\Delta \vdash (nth\ k\ \gamma_1);(nth\ k\ \gamma_2)$	$\rightsquigarrow$	$nth\ k\ (\gamma_1;\gamma_2)$	when $\Delta \vdash^{\text{co}} \gamma_1;\gamma_2 : \sigma_1 \sim_{\#} \sigma_2$

Figure 7 Coercion simplification (I).

4.1.1 Pulling reflexivity up

Rules ReflApp, ReflAll, ReflElimL, and ReflElimR, deal with uses of reflexivity. Rules ReflApp and ReflAll "swallow" constructors from the coercion language (coercion application, and quantification respectively) into the type language (type application, and quantification respectively). Hence they pull reflexivity as high as possible in the tree structure of a coercion term. Rules ReflElimL and ReflElimR simply eliminate reflexivity uses that are composed with other coercions.

4.1.2 Pushing symmetry down

Uses of symmetry, contrary to reflexivity, are pushed as close to the leaves as possible or eliminated, (rules SymRefl, SymAll, SymApp, SymTrans, and SymSym) only getting stuck at terms of the form $sym\ x$ and $sym\,(C\ \overline{\gamma})$. The idea is that by pushing uses of symmetry towards the leaves, the rest of the rules may completely ignore symmetry, except where symmetry-pushing gets stuck (variables or axiom applications).

4.1.3 Reducing coercions

Rules RedNth, RedInstCo, and RedInstTy comprise the first interesting group of rules. They eliminate uses of injectivity and instantiation. Rule RedNth is concerned with the case where we wish to decompose a coercion of type $H\ \overline{\phi} \sim_{\#} H\ \overline{\sigma}$, where the coercion term contains H in its head. Notice that H is a type and may already be applied to some type arguments $\overline{\tau}^{1..\ell}$, and hence the rule has to account for selection from the first ℓ arguments, or a later argument. Rule RedInstCo deals with instantiation of a polymorphic coercion with a type. Notice that in rule RedInstCo the quantified variable may only appear "protected" under some $\langle\sigma\rangle$ inside γ, and hence simply substituting $\gamma[\phi/a]$ is guaranteed to produce a syntactically well-formed coercion. Rule RedInstTy deals with the instantiation of a polymorphic coercion that is *just* a type.

4.1.4 Eta expanding and subsequent reducing

Redexes of RedNth and RedInstCo or RedInstTy may not be directly visible. Consider $nth\ k\ (\langle H\ \overline{\tau}^{1..\ell}\rangle\ \overline{\gamma}; \gamma)$. The use of transitivity stands in our way for the firing of rule RedNth. To the rescue, rules EtaAllL, EtaAllR, EtaNthL, and EtaNthR, push decomposition or instantiation through transitivity and eliminate such redexes. We call these rules "eta" because in effect we are η-expanding and immediately reducing one of the components of the transitive composition. Here is a decomposition of EtaAllL in smaller steps that involve an η-expansion (of γ_2 in the second line):

$$
\begin{array}{ll}
 & ((\forall a{:}\eta\,.\,\gamma_1); \gamma_2)@\phi \\
\rightsquigarrow & ((\forall a{:}\eta\,.\,\gamma_1); (\forall a{:}\eta\,.\,\gamma_2@a))@\phi \\
\rightsquigarrow & (\forall a{:}\eta\,.\,\gamma_1; \gamma_2@a)@\phi \quad \rightsquigarrow \quad \gamma_1[\phi/a]; \gamma_2@\phi
\end{array}
$$

We have merged these steps in a single rule to facilitate the proof of termination. In doing this, we do not lose any reactions, since all of the intermediate terms can also reduce to the final coercion.

There are many design possibilities for rules that look like our η-rules. For instance one may wonder why we are not always expanding terms of the form $\gamma_1; (\forall a{:}\eta\,.\,\gamma_2)$ to $\forall a{:}\eta\,.\,\gamma_1@a; \gamma_2$, whenever γ_1 is of type $\forall a{:}\eta\,.\,\tau \sim_{\#} \forall a{:}\eta\,.\,\phi$. We experimented with several

$$\frac{\Delta \vdash^{\text{co}} c : \tau \sim_{\#} \upsilon}{\Delta \vdash c; sym\ c \rightsquigarrow \langle \tau \rangle}\ \textsc{VarSym} \qquad \frac{\Delta \vdash^{\text{co}} c : \tau \sim_{\#} \upsilon}{\Delta \vdash sym\ c; c \rightsquigarrow \langle \upsilon \rangle}\ \textsc{SymVar}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\upsilon)}{\Delta \vdash C\ \overline{\gamma}_1; sym\,(C\ \overline{\gamma}_2) \rightsquigarrow [\overline{a} \mapsto \overline{\gamma_1; sym\ \gamma_2}]\uparrow(\tau)}\ \textsc{AxSym} \qquad \frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\tau)}{\Delta \vdash sym\,(C\ \overline{\gamma}_1); C\ \overline{\gamma}_2 \rightsquigarrow [\overline{a} \mapsto \overline{sym\ \gamma_1; \gamma_2}]\uparrow(\upsilon)}\ \textsc{SymAx}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\upsilon) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\upsilon)}{\Delta \vdash (C\ \overline{\gamma}_1); \delta \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \textsc{AxSuckR} \qquad \frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\tau) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_1]\uparrow(\tau)}{\Delta \vdash \delta; (C\ \overline{\gamma}_2) \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \textsc{AxSuckL}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\tau) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\tau)}{\Delta \vdash sym\,(C\ \overline{\gamma}_1); \delta \rightsquigarrow sym\,(C\ \overline{sym\ \gamma_2;\gamma_1})}\ \textsc{SymAxSuckR}$$

$$\frac{(C\ \overline{(a{:}\eta)} : \tau \sim_{\#} \upsilon) \in \Gamma \quad \overline{a} \subseteq ftv(\upsilon) \quad nontriv(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_1]\uparrow(\upsilon)}{\Delta \vdash \delta; sym\,(C\ \overline{\gamma}_2) \rightsquigarrow sym\,(C\ \overline{\gamma_2; sym\ \gamma_1})}\ \textsc{SymAxSuckL}$$

Figure 8 Coercion simplification (II).

variations like this, but we found that such expansions either complicated the termination argument, or did not result in smaller coercion terms. Our rules in effect perform η-expansion *only* when there is a firing reduction directly after the expansion.

4.1.5 Pushing transitivity down

Rules PushApp, PushAll, PushNth, and PushInst push uses of transitivity *down* the structure of a coercion term, towards the leaves. These rules aim to reveal more redexes at the leaves, that will be reduced by the next (and final) set of rules. Notice that rules PushInst and PushNth impose side conditions on the transitive composition $\gamma_1;\gamma_2$. Without these conditions, the resulting coercion may not be well-formed. Take $\gamma_1 = \forall a{:}\eta.\langle T\ a\ a\rangle$ and $\gamma_2 = \forall a{:}\eta.\langle T\ a\ \texttt{Int}\rangle$. It is certainly the case that $(\gamma_1@\texttt{Int});(\gamma_2@\texttt{Int})$ is well formed. However, $\vdash^{\text{co}} \gamma_1 : \forall a{:}\eta.\ T\ a\ a \sim_{\#} \forall a{:}\eta.\ T\ a\ a$ and $\vdash^{\text{co}} \gamma_2 : \forall a{:}\eta.\ T\ a\ \texttt{Int} \sim_{\#} \forall a{:}\eta.\ T\ a\ \texttt{Int}$, and hence $(\gamma_1;\gamma_2)@\texttt{Int}$ is not well-formed. A similar argument applies to rule PushNth.

4.1.6 Leaf reactions

When transitivity and symmetry have been pushed as low as possible, new redexes may appear, for which we introduce rules VarSym, SymVar, AxSym, SymAx, AxSuckR, AxSuckL, SymAxSuckR, SymAxSuckL. (Figure 8)

- Rules VarSym and SymVar are entirely straightforward: a coercion variable (or its symmetric coercion) meets its symmetric coercion (or the variable) and the result is the identity.
- Rules AxSym and SymAx are more involved. Assume that the axiom $(C\ (\overline{a{:}\eta}){:}\tau \sim_{\#} \upsilon) \in \Gamma$, and a well-formed coercion of the form: $C\ \overline{\gamma}_1; sym\,(C\ \overline{\gamma}_2)$. Moreover $\Delta \vdash^{\text{co}}$

$$\boxed{[a \mapsto \gamma]\uparrow(\tau) = \gamma'}$$

$$
\begin{array}{lcl}
[a \mapsto \gamma]\uparrow(a) & = & \gamma \\
[a \mapsto \gamma]\uparrow(b) & = & \langle b \rangle \\
[a \mapsto \gamma]\uparrow(H) & = & \langle H \rangle \\
[a \mapsto \gamma]\uparrow(F) & = & \langle F \rangle \\
[a \mapsto \gamma]\uparrow(\tau_1\ \tau_2) & = & \begin{cases} \langle \phi_1\ \phi_2 \rangle \text{ when } [a \mapsto \gamma]\uparrow(\tau_i) = \langle \phi_i \rangle \\ ([a \mapsto \gamma]\uparrow(\tau_1))\ ([a \mapsto \gamma]\uparrow(\tau_2)) \text{ otherwise} \end{cases} \\
[a \mapsto \gamma]\uparrow(\forall b{:}\eta\,.\,\tau) & = & \begin{cases} \langle \forall a{:}\eta\,.\,\phi \rangle \text{ when } [a \mapsto \gamma]\uparrow(\tau) = \langle \phi \rangle \\ \forall b{:}\eta\,.([a \mapsto \gamma]\uparrow(\tau)) \text{ otherwise} \quad (b \notin \mathit{ftv}(\gamma), b \neq a) \end{cases}
\end{array}
$$

Figure 9 Lifting.

$\overline{\gamma}_1 : \overline{\sigma}_1 \sim_\# \overline{\phi}_1$ and $\Delta \vdash^{\text{co}} \overline{\gamma}_2 : \overline{\sigma}_2 \sim_\# \overline{\phi}_2$. Then we know that $\Delta \vdash^{\text{co}} C\ \overline{\gamma}_1; \mathit{sym}\,(C\ \overline{\gamma}_2) : \tau[\overline{\sigma}_1/\overline{a}] \sim_\# \tau[\overline{\sigma}_2/\overline{a}]$. Since the composition is well-formed, it must be the case that $\upsilon[\overline{\phi}_1/\overline{a}] = \upsilon[\overline{\phi}_2/\overline{a}]$. If $\overline{a} \subseteq \mathit{ftv}(\upsilon)$ then it must be $\overline{\phi}_1 = \overline{\phi}_2$. Hence, the pointwise composition $\overline{\gamma_1; \mathit{sym}\ \gamma_2}$ is well-formed and of type $\overline{\sigma}_1 \sim_\# \overline{\sigma}_2$. Consequently, we may replace the original coercion with the *lifting* of τ over a substitution that maps $\overline{a}$ to $\overline{\gamma_1; \mathit{sym}\ \gamma_2}$: $[\overline{a} \mapsto \overline{\gamma_1; \mathit{sym}\ \gamma_2}]\uparrow(\tau)$.

What is this lifting operation, of a substitution from type variables to coercions, over a type? Its result is a new coercion, and the definition of the operation is given in Figure 9. The easiest way to understand it is by its effect on a type:

▶ **Lemma 3** (Lifting). *If* $\Delta, (a{:}\eta) \vdash^{\text{ty}} \tau : \eta$ *and* $\Delta \vdash^{\text{co}} \gamma : \sigma \sim \phi$ *such that* $\Delta \vdash^{\text{ty}} \sigma : \eta$ *and* $\Delta \vdash^{\text{ty}} \phi : \eta$, *then* $\Delta \vdash^{\text{co}} [a \mapsto \gamma]\uparrow(\tau) : \tau[\sigma/a] \sim_\# \tau[\phi/a]$

Notice that we have made sure that lifting pulls reflexivity as high as possible in the syntax tree – the only significance of this on-the-fly normalization was that it appeared to simplify the argument we have given for termination of coercion normalization.

Returning to rules AxSym and SymAx, we stress that the side condition is essential for the rule to be sound. Consider the following example:

$$C(a{:}\star) : F\ [a] \sim_\# \texttt{Int} \in \Gamma$$

Then $(C\ \langle \texttt{Int} \rangle); \mathit{sym}\,(C\ \langle \texttt{Bool} \rangle)$ is well-formed and of type $F\ [\texttt{Int}] \sim_\# F\ [\texttt{Bool}]$, but $\langle F \rangle\ (\langle \texttt{Int} \rangle; \mathit{sym}\,\langle \texttt{Bool} \rangle)$ is not well-formed! Rule SymAx is symmetric and has a similar soundness side condition on the free variables of τ this time.

The rest of the rules deal with the case when an axiom meets a lifted type – the reaction swallows the lifted type inside the axiom application. For instance, here is rule AxSuckR:

$$\frac{(C\ (\overline{a{:}\eta}){:}\tau \sim_\# \upsilon) \in \Gamma \quad \overline{a} \subseteq \mathit{ftv}(\upsilon) \quad \mathit{nontriv}(\delta) \quad \delta = [\overline{a} \mapsto \overline{\gamma}_2]\uparrow(\upsilon)}{\Delta \vdash (C\ \overline{\gamma}_1); \delta \rightsquigarrow C\ \overline{\gamma_1;\gamma_2}}\ \text{AxSuckR}$$

This time let us assume that $\Delta \vdash^{\text{co}} \overline{\gamma}_1 : \overline{\sigma}_1 \sim_\# \overline{\phi}_1$. Consequently $\Delta \vdash^{\text{co}} C\ \overline{\gamma}_1 : \tau[\overline{\sigma}_1/\overline{a}] \sim_\# \upsilon[\overline{\phi}_1/\overline{a}]$. Since $\overline{a} \subseteq \mathit{ftv}(\upsilon)$ it must be that $\Delta \vdash^{\text{co}} \overline{\gamma}_2 : \overline{\phi}_1 \sim_\# \overline{\phi}_3$ for some $\overline{\phi}_3$ and we can pointwise compose $\overline{\gamma_1;\gamma_2}$ to get coercions between $\overline{\sigma}_1 \sim_\# \overline{\phi}_3$. The resulting coercion $C\ \overline{\gamma_1;\gamma_2}$ is well-formed and of type $\tau[\overline{\sigma}_1/\overline{a}] \sim_\# \upsilon[\overline{\phi}_3/\overline{a}]$. Rules AxSuckL, SymAxSuckL, and SymAxSuckR involve a similar reasoning.

The side condition $\mathit{nontriv}(\delta)$ is not restrictive in any way – it merely requires that

δ contains some variable c or axiom application. If not, then δ can be converted to reflexivity:

▶ **Lemma 4.** *If* $\vdash^{\infty} \delta : \sigma \sim_{\#} \phi$ *and* $\neg$ *nontriv*(δ)*, then* $\delta \longrightarrow^* \langle \phi \rangle$.

Reflexivity, when transitively composed with any other coercion, is eliminable via ReflElimL/R or and consequently the side condition is not preventing any reactions from firing. It will, however, be useful in the simplification termination proof in Section 6.

The purpose of rules AxSuckL/R and SymAxSuckL/R is to eliminate intermediate coercions in a big transitive composition chain, to give the opportunity to an axiom to meet its symmetric version and react with rules AxSym and SymAx. In fact this rule is *precisely* what we need for the impressive simplifications from Section 3. Consider that example again:

$$
\begin{array}{rcll}
\gamma_5 & = & \gamma_2;\gamma_3;\gamma_4 & \\
 & = & sym\,(C_N\ \langle\tau_1\rangle);(\langle N\rangle\ \gamma_1);(C_N\ \langle\tau_2\rangle) & (\text{AxSucL with } \delta := (\langle N\rangle\ \gamma_1)) \\
 & \longrightarrow & sym\,(C_N\ \langle\tau_1\rangle);(C_N\ (\gamma_1;\langle\tau_2\rangle)) & (\text{ReflElimR with } \gamma := \gamma_1, \phi := \tau_2) \\
 & \longrightarrow & sym\,(C_N\ \langle\tau_1\rangle);(C_N\ \gamma_1) & (\text{SymAx}) \\
 & \longrightarrow & \langle\rightarrow\rangle\ (\langle\tau_1\rangle;\gamma_1)\ \langle \texttt{Int}\rangle & (\text{ReflElimL with } \phi := \tau_1, \gamma := \gamma_1) \\
 & \longrightarrow & \langle\rightarrow\rangle\ \gamma_1\ \langle \texttt{Int}\rangle &
\end{array}
$$

Notably, rules AxSuckL/R and SymAxSuckL/R generate axiom applications of the form $C\ \overline{\gamma}$ (with a coercion as argument). In our previous papers, the syntax of axiom applications was $C\ \overline{\tau}$, with *types* as arugments. But we need the additional generality to allow coercions rewriting to proceed without getting stuck.

5 Coercion simplification in GHC

To assess the usefulness of coercion simplification we added it to GHC. For Haskell programs that make no use of GADTs or type families, the effect will be precisely zero, so we took measurements on two bodies of code. First, our regression suite of 151 tests for GADTs and type families; these are all very small programs. Second, the `Data.Accelerate` library that we know makes use of type families [5]. This library consists of 18 modules, containing 8144 lines of code.

We compiled each of these programs with and without coercion simplification, and measured the percentage reduction in size of the coercion terms with simplification enabled. This table shows the minimum, maximum, and aggregate reduction, taken over the 151 tests and 18 modules respectively. The "aggregate reduction" is obtained by combining all the programs in the group (testsuite or `Accelerate`) into one giant "program", and computing the reduction in coercion size.

	Testsuite	Accelerate
Minimum	−97%	−81%
Maximum	+14%	0%
Aggregate	**−58%**	**−69%**

There is a substantial aggregate decrease of 58% in the testsuite and 69% in `Accelerate`, with a massive 97% decrease in special cases. These special cases should not be taken lightly: in one program the types and coercions taken together were five times bigger than the term they decorated; after simplification they were "only" twice as big. The coercion simplifier makes the compiler less vulnerable to falling off a cliff.

Only one program showed an increase in coercion size, of 14%, which turned out to be the effect of this rewrite:

$$sym\,(C;D) \quad \longrightarrow \quad (sym\ D);(sym\ C)$$

Smaller coercion terms make the compiler faster, but the normalization algorithm itself consumes some time. However, the effect on compile time is barely measurable (less than 1%), and we do not present detailed figures.

Of course none of this would matter if coercions were always tiny, so that they took very little space in the first place. And indeed that is often the case. But for programs that make heavy use of type functions, un-optimised coercions can dominate compile time. For example, the `Accelerate` library makes heavy use of type functions. The time and memory consumption of compiling all 21 modules of the library are as follows:

	Compile time	Memory allocated	Max residency
With coercion optimisation	68s	31 *Gbyte*	153 *Mbyte*
Without coercion optimisation	291s	51 *Gbyte*	2,000 *Mbyte*

As you can see, the practical effects can be extreme; the cliff is very real.

6 Termination and confluence

We have demonstrated the effectiveness of the algorithm in practice, but we must also establish termination. This is important, since it would not be acceptable for a compiler to loop while simplifying a coercion, no matter what axioms are declared by users. Since the rules fire non-deterministically, and some of the rules (such as RedInstCo or AxSym) create potentially larger coercion trees, termination is not obvious.

6.1 Termination

To formalize a termination argument, we introduce several definitions in Figure 10. The *axiom polynomial* of a coercion over a distinguished variable z, $p(\cdot)$, returns a polynomial with natural number coefficients that can be compared to any other polynomial over z. The *coercion weight* of a coercion is defined as the function $w(\cdot)$ and the *symmetry weight* of a coercion is defined with the function $sw(\cdot)$ in Figure 10. Unlike the polynomial and coercion weights of a coercion, $sw(\cdot)$ does take symmetry into account. Finally, we will also use the *number of coercion applications and coercion* $\forall$*-introductions*, denoted with $intros(\cdot)$ in what follows.

Our termination argument comprises of the lexicographic left-to-right ordering of:

$$\mu(\cdot) = \langle p(\cdot), w(\cdot), intros(\cdot), sw(\cdot)\rangle$$

We will show that each of the $\rightsquigarrow$ reductions reduces this tuple. For this to be a valid termination argument for $(\longrightarrow)$ we need two more facts about *each* component measure, namely that (i) $(=)$ and $(<)$ are preserved under arbitrary contexts, and (ii) each component is invariant with respect to the associativity of $(;)$.

▶ **Lemma 5.** *If* $\Delta \vdash^{\infty} \gamma_1 : \tau \sim_{\#} \sigma$ *and* $\gamma_1 \cong \gamma_2$ *modulo associativity of* $(;)$*, then* $p(\gamma_1) = p(\gamma_2)$*,* $w(\gamma_1) = w(\gamma_2)$*,* $intros(\gamma_1) = intros(\gamma_2)$*, and* $sw(\gamma_1) = sw(\gamma_2)$*.*

Proof. This is a simple inductive argument, the only interesting case is the case for $p(\cdot)$ where the reader can calculate that $p(\gamma_1;(\gamma_2;\gamma_3)) = p((\gamma_1;\gamma_2);\gamma_3)$ and by induction we are done. ◀

Axiom polynomial

$$
\begin{array}{lcl}
p(sym\ \gamma) & = & p(\gamma) \\
p(C\ \overline{\gamma}) & = & z \cdot \Sigma p(\gamma_i) + z + 1 \\
p(c) & = & 1 \\
p(\gamma_1;\gamma_2) & = & p(\gamma_1) + p(\gamma_2) + p(\gamma_1) \cdot p(\gamma_2) \\
p(\langle\phi\rangle) & = & 0 \\
p(nth\ k\ \gamma) & = & p(\gamma) \\
p(\gamma@\phi) & = & p(\gamma) \\
p(\gamma_1\ \gamma_2) & = & p(\gamma_1) + p(\gamma_2) \\
p(\forall a{:}\eta.\gamma) & = & p(\gamma)
\end{array}
$$

Coercion weight

$$
\begin{array}{lcl}
w(sym\ \gamma) & = & w(\gamma) \\
w(C\ \overline{\gamma}) & = & \Sigma w(\gamma_i) + 1 \\
w(c) & = & 1 \\
w(\gamma_1;\gamma_2) & = & 1 + w(\gamma_1) + w(\gamma_2) \\
w(\langle\phi\rangle) & = & 1 \\
w(nth\ k\ \gamma) & = & 1 + w(\gamma) \\
w(\gamma@\phi) & = & 1 + w(\gamma) \\
w(\gamma_1\ \gamma_2) & = & 1 + w(\gamma_1) + w(\gamma_2) \\
w(\forall a{:}\eta.\gamma) & = & 1 + w(\gamma)
\end{array}
$$

Symmetry weight

$$
\begin{array}{lcl}
sw(sym\ \gamma) & = & w(\gamma) + sw(\gamma) \\
sw(C\ \overline{\gamma}) & = & \Sigma sw(\gamma_i) \\
sw(c) & = & 0 \\
sw(\gamma_1;\gamma_2) & = & sw(\gamma_1) + sw(\gamma_2) \\
sw(\langle\phi\rangle) & = & 0 \\
sw(nth\ k\ \gamma) & = & sw(\gamma) \\
sw(\gamma@\phi) & = & sw(\gamma) \\
sw(\gamma_1\ \gamma_2) & = & sw(\gamma_1) + sw(\gamma_2) \\
sw(\forall a{:}\eta.\gamma) & = & sw(\gamma)
\end{array}
$$

Figure 10 Metrics on coercion terms.

▶ **Lemma 6.** *If* $\Gamma, \Delta \vDash^{co} \gamma_i : \tau \sim_\# \sigma$ *(for* $i = 1, 2$*) and* $p(\gamma_1) < p(\gamma_2)$ *then* $p(\mathcal{G}[\gamma_1]) < p(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{co} \mathcal{G}[\gamma_i] : \phi \sim_\# \phi'$*. Similarly if we replace* $(<)$ *with* $(=)$*.*

Proof. By induction on the shape of $\mathcal{G}$. The only interesting case is the transitivity case again. Let $\mathcal{G} = \gamma; \mathcal{G}'$. Then $p(\gamma; \mathcal{G}'[\gamma_1]) = p(\gamma) + p(\mathcal{G}'[\gamma_1]) + p(\gamma) \cdot p(\mathcal{G}'[\gamma_1])$ whereas $p(\gamma; \mathcal{G}'[\gamma_2]) = p(\gamma) + p(\mathcal{G}'[\gamma_2]) + p(\gamma) \cdot p(\mathcal{G}'[\gamma_2])$. Now, either $p(\gamma) = 0$, in which case we are done by induction hypothesis for $\mathcal{G}'[\gamma_1]$ and $\mathcal{G}'[\gamma_2]$, or $p(\gamma) \neq 0$ in which case again induction hypothesis gives us the result since we are multiplying $p(\mathcal{G}'[\gamma_1])$ and $p(\mathcal{G}'[\gamma_2])$ by the same polynomial. The interesting "trick" is that the polynomial for transitivity contains both the product of the components *and* their sum (since product alone is not preserved by contexts!). ◀

▶ **Lemma 7.** *If* $\Gamma, \Delta \vDash^{co} \gamma_i : \tau \sim_\# \sigma$ *and* $w(\gamma_1) < w(\gamma_2)$ *then* $w(\mathcal{G}[\gamma_1]) < w(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{co} \mathcal{G}[\gamma_i] : \phi \sim_\# \phi'$*. Similarly if we replace* $(<)$ *with* $(=)$*.*

▶ **Lemma 8.** *If* $\Gamma, \Delta \vDash^{co} \gamma_i : \tau \sim_\# \sigma$ *and* $intros(\gamma_1) < intros(\gamma_2)$ *then* $intros(\mathcal{G}[\gamma_1]) < intros(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{co} \mathcal{G}[\gamma_i] : \phi \sim_\# \phi'$*. Similarly if we replace* $(<)$ *with* $(=)$*.*

▶ **Lemma 9.** *If* $\Gamma, \Delta \vDash^{co} \gamma_i : \tau \sim_\# \sigma$*,* $w(\gamma_1) \leq w(\gamma_2)$*, and* $sw(\gamma_1) < sw(\gamma_2)$ *then* $sw(\mathcal{G}[\gamma_1]) < sw(\mathcal{G}[\gamma_2])$ *for any* $\mathcal{G}$ *with* $\Gamma \vDash^{co} \mathcal{G}[\gamma_i] : \phi \sim_\# \phi'$*.*

Proof. The only interesting case is when $\mathcal{G} = sym\ \mathcal{G}'$ and hence we have that $sw(\mathcal{G}[\gamma_1]) = sw(sym\ \mathcal{G}'[\gamma_1]) = w(\mathcal{G}'[\gamma_1]) + sw(\mathcal{G}'[\gamma_1])$. Similarly $sw(\mathcal{G}[\gamma_2]) = w(\mathcal{G}'[\gamma_2]) + sw(\mathcal{G}'[\gamma_2])$. By the precondition for the weights and induction hypothesis we are done. The precondition on the weights is not restrictive, since $w(\cdot)$ has higher precedence than $sw(\cdot)$ inside $\mu(\cdot)$. ◀

The conclusion is the following theorem.

▶ **Theorem 10.** *If* $\gamma \cong \mathcal{G}[\gamma_1]$ *modulo associativity of* $(;)$ *and* $\Delta \vDash^{co} \gamma_1 : \sigma \sim_\# \phi$*, and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *such that* $\mu(\gamma_2) < \mu(\gamma_1)$*, it is the case that* $\mu(\mathcal{G}[\gamma_2]) < \mu(\gamma)$*.*

▶ **Corollary 11.** $(\longrightarrow)$ *terminates on well-formed coercions if each of the* $\rightsquigarrow$ *transitions reduces* $\mu(\cdot)$.

Note that often the term rewrite literature requires similar conditions (preservation under contexts and associativity), but also *stability under substitution* (e.g. see [1], Chapter 5). In our setting, variables are essentially treated as constants and this is the reason that we do not rely on stability under substitutions. For instance the rule ReflElimR $\Delta|-\gamma;\langle\phi\rangle \rightsquigarrow \gamma$ is *not* expressed as $\Delta|-c;\langle\phi\rangle \rightsquigarrow c$, as would be customary in a more traditional term-rewrite system presentation.

We finally show that indeed each of the $\rightsquigarrow$ steps reduces $\mu(\cdot)$.

▶ **Theorem 12** (Termination). *If* $\Delta \vDash^{\infty} \gamma_1 : \sigma \sim_{\#} \phi$ *and* $\Delta \vdash \gamma_1 \rightsquigarrow \gamma_2$ *then* $\mu(\gamma_2) < \mu(\gamma_1)$.

Proof. It is easy to see that the reflexivity rules, the symmetry rules, the reduction rules, and the η-rules preserve or reduce the polynomial component $p(\cdot)$. The same is true for the push rules but the proof is slightly more interesting. Let us consider PushApp, and let us write p_i for $p(\gamma_i)$. We have that $p((\gamma_1\ \gamma_2);(\gamma_3\ \gamma_4)) = p_1+p_2+p_3+p_4+p_1p_3+p_2p_3+p_1p_4+p_2p_4$. On the other hand $p((\gamma_1;\gamma_3)\ (\gamma_2;\gamma_4)) = p_1+p_3+p_1p_3+p_2+p_4+p_2p_4$ which is a smaller or equal polynomial than the left-hand side polynomial. Rule PushAll is easier. Rules PushInst and PushNth have exactly the same polynomials on the left-hand and the right-hand side so they are ok. Rules VarSym and SymVar reduce $p(\cdot)$. The interesting bit is with rules AxSym, SymAx, and AxSuckR/L and SymAxSuckR/L. We will only show the cases for AxSym and AxSuckR as the rest of the rules involve very similar calculations:

- Case SymAx. We will use the notational convention $\overline{p}_1$ for $p(\overline{\gamma}_1)$ (a vector of polynomials) and similarly $\overline{p}_2$ for $p(\overline{\gamma}_2)$. Then the left-hand side polynomial is:

$$\begin{aligned}&(z\Sigma\overline{p}_1+z+1)+(z\Sigma\overline{p}_2+z+1)+\\ &\qquad(z\Sigma\overline{p}_1+z+1)\cdot(z\Sigma\overline{p}_2+z+1)=\\ &(z^2+2z)\Sigma\overline{p}_1+(z^2+2z)\Sigma\overline{p}_2+z^2\Sigma\overline{p}_1\Sigma\overline{p}_2+(z^2+4z+3)\end{aligned}$$

  For the right-hand side polynomial we know that each $\gamma_{1i};sym\ \gamma_{2i}$ will have polynomial $p_{1i}+p_{2i}+p_{1i}p_{2i}$ and it cannot be repeated inside the lifted type more than a finite number of times (bounded by the maximum number of occurrences of a type variable from $\overline{a}$ in type τ), call it k. Hence the right-hand side polynomial is smaller or equal to:

$$k\Sigma\overline{p}_1+k\Sigma\overline{p}_2+k\Sigma(p_{1i}p_{2i}) \leq k\Sigma\overline{p}_1+k\Sigma\overline{p}_2+k\Sigma\overline{p}_1\Sigma\overline{p}_2$$

  But that polynomial is strictly smaller than the left-hand side polynomial, hence we are done.
- Case AxSuckR. In this case the left-hand side polynomial is going to be greater or equal to (because of reflexivity inside δ and because some of the $\overline{a}$ variables may appear more than once inside υ it is not exactly equal to) the following:

$$\begin{aligned}&(z\Sigma\overline{p}_1+z+1)+\Sigma\overline{p}_2+(z\Sigma\overline{p}_1+z+1)\Sigma\overline{p}2=\\ &\qquad z\Sigma\overline{p}_1\Sigma\overline{p}_2+z\Sigma\overline{p}_1+z\Sigma\overline{p}_2+2\Sigma\overline{p}_2+z+1\end{aligned}$$

  On the other hand, the right-hand side polynomial is:

$$z\Sigma(p_{1i}+p_{2i}+p_{1i}p_{2i})+z+1 \ \leq\ z\Sigma\overline{p}_1+z\Sigma\overline{p}_2+z\Sigma\overline{p}_1\Sigma\overline{p}_2+z+1$$

  We observe that there is a difference of $2\Sigma\overline{p}_2$, but we know that δ satisfies $nontriv(\delta)$, and consequently there must exist some variable or axiom application inside one of the $\overline{\gamma}_2$. Therefore, $\Sigma\overline{p}_2$ is *non-zero* and the case is finished.

It is the arbitrary copying of coercions $\overline{\gamma}_1$ and $\overline{\gamma}_2$ in rules AxSym and SymAx that prevents simpler measures that only involve summation of coercions for axioms or transitivity. Other reasonable measures such as the height of transitivity uses from the leaves would not be preserved from contexts, due to AxSym again.

So far we've shown that all rules but the axiom rules preserve the polynomials, and the axiom rules reduce them. We next show that in the remaining rules, some other component reduces, lexicographically. Reflexivity rules reduce $w(\cdot)$. Symmetry rules preserve $w(\cdot)$ and $intros(\cdot)$ but reduce $sw(\cdot)$. Reduction rules and η-rules reduce $w(\cdot)$. Rules PushApp and PushAll preserve or reduce $w(\cdot)$ but certainly reduce $intros(\cdot)$. Rules PushInst and PushNth reduce $w(\cdot)$. ◀

We conclude that $(\longrightarrow)$ terminates.

6.2 Confluence

Due to the arbitrary types of axioms and coercion variables in the context, we do not expect confluence to be true. Here is a short example that demonstrates the lack of confluence; assume we have the following in our context:

$$C_1\,(a{:}\star \to \star) : F\ a \sim_\# a$$
$$C_2\,(a{:}\star \to \star) : G\ a \sim_\# a$$

Consider the coercion:

$$(C_1\ \langle\sigma\rangle); sym\,(C_2\ \langle\sigma\rangle)$$

of type $F\ \sigma \sim_\# G\ \sigma$. In one reduction possibility, using rule AxSuckR, we may get

$$C_1\ (sym\,(C_2\ \langle\sigma\rangle))$$

In another possibility, using SymAxSuckL, we may get

$$sym\,(C_2\ (sym\,(C_1\ \langle\sigma\rangle)))$$

Although the two normal forms are different, it is unclear if one of them is "better" than the other.

Despite this drawback, confluence or syntactic characterization of normal forms is, for our purposes, of secondary importance (if possible at all for open coercions in such an under-constrained problem!), since we never reduce coercions for the purpose of comparing their normal forms. That said, we acknowledge that experimental results may vary with respect to the actual evaluation strategy, but we do not expect wild variations.

7 Related and future work

Traditionally, work on proof theory is concerned with proof normalization theorems, namely cut-elimination. Category and proof theory has studied the commutativity of diagrams in *monoidal categories* [12], establishing coherence theorems. In our setting Lemma 4 expresses such a result: any coercion that does not include axioms or free coercion variables is equivalent to reflexivity. More work on proof theory is concerned with cut-elimination theorems – in our setting eliminating transitivity completely is plainly impossible due to the presence of axioms. Recent work on *2-dimensional type theory* [10] provides an equivalence relation on equality proofs (and terms), which suffices to establish that types enjoy canonical forms.

Although that work does not provide an algorithm for checking equivalence (this is harder to do because of actual computation embedded with isomorphisms), that definition shares many rules with our normalization algorithm. Finally there is a large literature in associative commutative rewrite systems [7, 2].

To our knowledge, most programming languages literature on coercions is not concerned with coercion simplification but rather with inferring the placement of coercions in source-level programs. Some recent examples are [11] and [17]. A comprehensive study of coercions *and their normalization* in programming languages is that of [8], motivated by coercion placement in a language with *type dynamic*. Henglein's coercion language differs to ours in that (i) coercions there are not symmetric, (ii) do not involve polymorphic axiom schemes and (iii) may have computational significance. Unlike us, Henglein is concerned with characterizations of minimal coercions and confluence, fixes an equational theory of coercions, and presents a normalization algorithm for that equational theory. In our case, in the absence of a denotational semantics for System FC and its coercions, such an axiomatization would be no more ad-hoc than the algorithm and hence not particularly useful: for instance we could consider adding type-directed equations like $\Delta \vdash \gamma \rightsquigarrow \langle \tau \rangle$ when $\Delta \vdash^{\text{co}} \gamma : \tau \sim_{\#} \tau$, or other equations that only hold in consistent or confluent axiom sets. It is certainly an interesting direction for future work to determine whether there even exists a maximal syntactic axiomatization of equalities between coercions with respect to some denotational semantics of System FC.

In the space of typed intermediate languages, xMLF[14] is a calculus with coercions that capture *instantiation* instead of equality, and which serves as target for the MLF language. Although the authors are not directly concerned with normalization as part of an intermediate language simplifier, their translation of the graph-based instantiation witnesses does produce xMLF normal proofs.

Finally, another future work direction would be to determine whether we can encode coercions as λ-terms, and derive coercion simplification by normalization in some suitable λ-calculus.

Acknowledgments

Thanks to Tom Schrijvers for early discussions and for contributing a first implementation. We would particularly like to thank Thomas Ströder for his insightful and detailed feedback in the run-up to submitting the final paper.

References

1 Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

2 Leo Bachmair and David A. Plaisted. Termination orderings for associative-commutative rewriting systems. *J. Symb. Comput.*, 1(4):329–349, December 1985.

3 Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP'05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM.

4 Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, 2005.

5 Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP'11, pages 3–14, New York, NY, USA, 2011. ACM.

6 James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

7 Nachum Dershowitz, Jien Hsiang, N. Alan Josephson, and David A. Plaisted. Associative-commutative rewriting. In *Proceedings of the Eighth international joint conference on Artificial intelligence – Volume 2*, IJCAI'83, pages 940–944, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

8 Fritz Henglein. Dynamic typing: syntax and proof theory. *Sci. Comput. Program.*, 22:197–230, June 1994.

9 Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In Cliff Jones and Bill Roscoe, editors, *Reflections on the work of CAR Hoare*. Springer, 2010.

10 Daniel R. Licata and Robert Harper. Canonicity for 2-dimensional type theory. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'12, pages 337–348, New York, NY, USA, 2012. ACM.

11 Zhaohui Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008.

12 Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

13 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP'06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.

14 Didier Rémy and Boris Yakobowski. A Church-style intermediate language for MLF. In Matthias Blume, Naoki Kobayashi, and German Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin / Heidelberg, 2010.

15 Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proc 4th International Workshop on Logical Frameworks and Meta-languages (LFM'04), Cork*, pages 106–124, July 2004.

16 Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI'07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, New York, NY, USA, 2007. ACM.

17 Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP'09, pages 329–340, New York, NY, USA, 2009. ACM.

18 Dimitrios Vytiniotis, Simon Peyton Jones, and Pedro Magalhaes. Equality proofs and deferred type errors. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*, pages 341–352, 2012.

19 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x): modular type inference with local assumptions. *Journal of Functional Programming*, 21, 2011.

20 Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'11, pages 227–240, New York, NY, USA, 2011. ACM.

Linear Logic and Strong Normalization

Beniamino Accattoli

Carnegie Mellon University – Pittsburgh, PA, USA, beniamino.accattoli@gmail.com

Abstract

Strong normalization for linear logic requires elaborated rewriting techniques. In this paper we give a new presentation of MELL proof nets, without any commutative cut-elimination rule. We show how this feature induces a compact and simple proof of strong normalization, via reducibility candidates. It is the first proof of strong normalization for MELL which does not rely on any form of confluence, and so it smoothly scales up to full linear logic. Moreover, it is an *axiomatic* proof, as more generally it holds for every set of rewriting rules satisfying three very natural requirements with respect to substitution, *commutation with promotion*, *full composition*, and *Kesner's IE property*. The insight indeed comes from the theory of explicit substitutions, and from looking at the exponentials as a substitution device.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.2 Semantics of Programming Languages

Keywords and phrases linear logic, proof nets, strong normalization, explicit substitutions

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.39

1 Introduction

Normalization is probably the most studied topic in proof theory and in the theory of functional programming languages, not to speak of rewriting theory. It comes in two flavors, *weak* or *strong* normalization. Weak normalization (WN) holds when there is an evaluation sequence reaching a normal form. This is the relevant notion of normalization, since for instance in proof theory it suffices to establish the completeness of the cut-free sub-system. Strong normalization (SN) is the variant where *all* evaluation sequences terminate, and it is mandatory when one is interested in exploring and comparing different evaluation strategies.

Sometimes, the gap between proving weak or strong normalization is minimal; for instance for the simply typed λ-calculus there is an extremely short argument for SN, due to van Daalen [34]. Some other times the gap is huge; *e.g.* for λ-calculi with explicit substitution (ES) [20] or for linear logic proof nets [28, 29]. The occasional increment of difficulty for SN is apparently related to the combinatorics of the rewriting system, independently of the logic/type system. This point of view seems to be justified by two facts. First, the proofs of SN for ES-calculi are usually much harder than for λ-calculus, even if the underlying type system is kept unchanged[1]. Second, the only proof method for proving SN for full linear logic starts by proving WN and then obtains SN using the conservation theorem[2], whose proof relies on a special form of local confluence and that holds even in untyped proof nets/calculi

[1] Most of the time one shows preservation of SN (PSN), *i.e.* that if t is SN with respect to β then it is SN with respect to the ES-calculus X under analysis, rather than SN for a fixed type system. The reason is that PSN is an untyped property which implies SN for X with respect to any typing discipline for which the λ-calculus is SN. Proving PSN or SN for a type system requires the same combinatorial reasoning, but PSN is a more general formulation, independent from the type system.

[2] The conservation theorem says that any term/net which is weakly normalizing for non-erasing evaluation steps is strongly normalizing for non-erasing evaluation steps.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 39–54

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

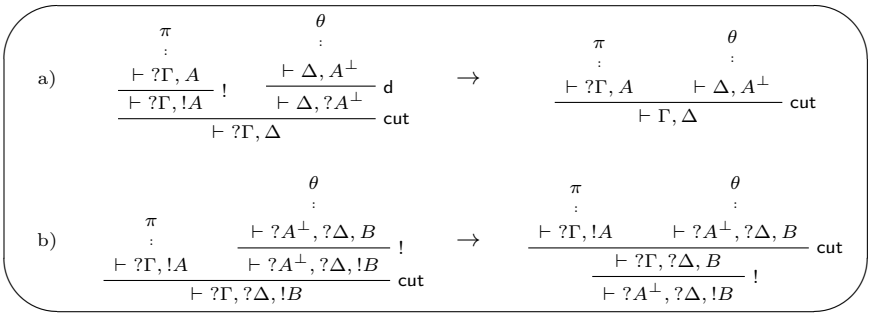


Figure 1 a) Principal case of cut-elimination; b) Commutative case of cut-elimination.

[28, 29]; this proof technique is sometimes called Gandy's method [12, 15] or Nederpelt's method [25, 33].

Proof nets and explicit substitutions both decompose evaluation in small steps. It is then generally believed that the gap in technical efforts between WN and SN is due to the granularity of the rules; the more evaluation is decomposed the harder is the proof of SN. In particular, these rewriting systems lack *orthogonality*, which is sometimes recognized as the reason behind the difficulty [29]. Here we show that this is a misleading point of view. The feature inducing many complications is rather the presence of commutative rewriting rules. We prove this fact by exhibiting a small-step and non-orthogonal presentation of MELL proof nets without any commutative cut-elimination rule, and enjoying a compact, modular, and informative proof of SN.

Commutations and boxes. Any proof of normalization in sequent calculus deals with two kinds of cases, the principal (or key) cases and the commutative cases. Principal cases arise when the last rules of the two cut proofs are those introducing (or contracting) the cut formulas, for instance as in Figure 1.a. The pattern is then replaced by a simpler one where the two last rules have been removed; these are the cases where something logical decreases. A commutative case instead arises when the last rule of one of the two cut proofs is not the one introducing the cut formula, as in Figure 1.b. In these cases no rule is removed. The rewriting consists only in re-arranging the structure of the proof, *i.e.* in *commuting* the cut upwards, in order to get closer to a principal case.

In the study of WN, commutative cases are certainly annoying—because they take most of the proof[3]—but they can be handled without too many efforts. When studying SN, instead, they become a serious obstacle. In sequent calculus the definition of a strongly normalizing cut-elimination is not evident, because some commutations (for instance of cut with itself) have to be taken as equivalences, otherwise the system has silly diverging reductions. This requires to switch to rewriting modulo, which is quite more technical than plain rewriting.

In his seminal paper on linear logic [14], Girard introduced proof nets, a graphical syntax alternative to sequent calculus. In proof nets deductive rules are disposed on the plane, in parallel, and connected only by their causal relation. There is no *last rule*, and so most commutative cut-elimination cases simply disappear. Unfortunately, to handle the exponentials Girard was forced to introduce *boxes*. They come with the *black-box principle*: "boxes are treated in a perfectly modular way: we can use the box B without knowing its

[3] Usually, in a system with n rules, each rule gives rise to at most 3 principal cases and at least $n-4$ commutative cases. Then among the $O(n^2)$ cases to consider, there are only $O(n)$ principal case but $O(n^2)$ commutative cases.

contents, *i.e.*, another box B' with exactly the same doors would do as well" [14].

According to this principle, boxes forbid interaction between their content and their outer environment. Proof nets have a dedicated commutative rule which brings a box inside another one (rule $\to_{\boxdot}$ in Figure 4, corresponding to Figure 1.b). Despite having brought a whole bunch of new perspectives and results on cut-elimination, for instance with respect to optimal reductions [5] or implicit computational complexity [16], proof nets have somehow failed in the original intent of simplifying the study of SN: it took 23 years to obtain a complete proof for full linear logic, and the outcome—due to Pagani and Tortora de Falco—requires sophisticated rewriting techniques [28], which are out of scope for most people without a special background in rewriting and proof nets. Moreover, despite the merit of filling an embarrassing hole in the literature (and of the impressive efforts it required), that proof is technically unsatisfactory as it relies on a special form of local confluence, while termination and confluence are independent properties[4].

In this paper we present an alternative approach, based on the removal of the black-box principle. Boxes are seen as decorations which do not prevent rules to interact through their borders. In some sense this is not a novelty, as a similar approach is taken in classic references on proof nets as [32, 8], where there is an exponential cut-elimination rule mimicking substitution in the λ-calculus. However, what is original here is that we pair this *box-crossing principle* with small-step rules. Following the new principle, our system has no commutative rule. At first sight it may seem to be only a minor variation, but we show that this change has surprising and impressive consequences on the proof of SN.

The rewriting technique. We prove SN using Girard's reducibility candidates (in the biorthogonal form, as first introduced in [14]), which are the only known technique for SN in a second order setting[5]. We abstract the proof with respect to any set of rewriting rules enjoying three natural properties, two of which are borrowed from the theory of explicit substitutions. We define a notion of (implicit) substitution for proof nets (noted $P\{x/Q\}$), and a notion of explicit substitution (noted $P[x/Q]$), which are essentially given by a big-step exponential rule and by an ordinary exponential cut, respectively. Then, we prove SN for every rewriting relation $\to$ enjoying:

1. *Commutation of substitution and promotion via* $\to$: namely $!(P\{x/Q\}) \to^* (!P)\{x/Q\}$ must hold. This property is specific to proof nets, and essentially corresponds to the permutation of contractions and weakenings with boxes; these premutations are necessary for the representation of any term calculus and so they are extremely natural.
2. *Full composition*: every explicit substitution can be evaluated fully, and independently by any other one, *i.e.* $P[x/Q] \to^* P\{x/Q\}$. This property is borrowed from the theory of explicit substitutions, and expresses a form of *context-freeness* for the evaluation of explicit substitution.
3. *Kesner's IE property [21]*: strong normalization is preserved by expanding *I*mplicit substitutions into *E*xplicit substitutions, namely if $P\{x/Q\} \in SN_\to$ and $Q \in SN_\to$ then $P[x/Q] \in SN_\to$. For ES-calculi this property holds without any typing assumption (proof nets are typed, but the proof of the IE property does not rely on types), and encapsulates the combinatorial content of the strong normalization argument.

[4] Sequent calculus LK for classical logic is strongly normalizing without being confluent, and λ-calculus is confluent without being even weakly normalizing.

[5] Second order quantifiers are here omitted. The reason is that the difficulties for SN in linear logic are related to the exponentials and not to the quantifiers; in fact—once one embraces reducibility candidates—the treatment of the quantifiers is smooth, and actually forces to deal with some details which make the proof less readable.

Delia Kesner has shown that preservation of strong normalization for ES-calculi can be reduced to the IE property. Such a property is the adaptation to ES of a classic expansion lemma in λ-calculus, called *the fundamental lemma of perpetuality* in [37]: if $t\{x/s\}u_1 \dots u_n \in SN_\beta$ and $s \in SN_\beta$ then $(\lambda x.t)su_1 \dots u_n \in SN_\beta$. This lemma holds in the untyped case, and it is the crucial clause in the inductive definition of strongly normalizing λ-terms found independently by van Raamsdonk and Severi [36] and Loader [23]. It appears in most SN arguments, in particular in Girard's proof for System F ([17], p. 44) or in van Daalen's short proof cited above. Essentially, IE reduces SN to an inverse preservation property with respect to substitution. The complexity of a proof of SN, then, depends on how nicely the rewriting rules interact with substitution.

Summing up, we generalize Kesner's technique to linear logic proof nets, a setting quite richer than λ-calculus, and isolate the role it plays in the reducibility candidates technique. In our commutation-free proof nets the IE property enjoys a simple and direct proof by induction on a triple. In contrast, in presence of commutations the induction does not go through, because commutative rules interfere with substitution, breaking the inductive invariant (see the last paragraph of Section 3).

Our proof can be seen as the equivalent for proof nets of the short proof of SN by van Daalen for the simply typed λ-calculus, but powered to higher-order linear logic via reducibility candidates. It is the first proof not relying on any form of confluence, and this feature—in contrast to all other known proofs of SN for MELL—let it smoothly scale up to full linear logic: the main difficulty posed by the additives is their stubborn reticence to be confluent in presence of the exponentials (case in which Hughes and van Glabbeek's confluent approach to the additives [18] does not work).

Sometimes, short proofs are not necessarily clear or informative. On the contrary, we believe that the main contribution of this work is the isolation of general and clear properties responsible for SN, making the proof understandable to anyone with a minimum background on proof nets and reducibility candidates. Furthermore, it sheds a new light on the role of commutative rules, and thus its interest goes well beyond the particular case of linear logic.

Related work. The original proof of Girard in [14] depends on a crucial lemma (which plays the role of the IE property here and yet is different) which was not proved in that paper. Later, Danos proved the lemma for second order MELL [7], but the proof is much less direct than the one we give here for the IE property. For MELL there is a proof of SN by Joinet [19], later refined by van Raamsdonk [35], but it does not scale up to second order. The first proof handling the additives is by Tortora de Falco and Pagani, in [28], and uses a conservation theorem. All these results rely on some form of confluence. For WN there also are a semantical proof by Okada [26] (see also the comments at page 53 in [28]) and an elegant formalized proof by Pfenning [31, 30]. A conservation theorem has also been used to prove SN for differential linear logic, by combining the results of Pagani and Tranquilli [29] with those on WN by Pagani [27] (only propositional) or by Gimenez [13] (propositional, but using reducibility candidates). The proof of SN presented here builds on our previous joint work with Kesner [3], and the new presentation of proof nets is a reformulation with explicit boxes of what naturally arises with implicit boxes [2, 1] and corresponds to rewriting rules *at a distance* on term calculi [3]. Finally, in [6] Bonelli studies reducibility candidates and normalization for a system F with explicit substitutions.

Road map. Section 2 defines MELL proof nets. Section 3 introduces implicit and explicit substitutions, and discusses full composition and the IE property. Section 4 introduces the terminology for reducibility candidates and proves SN. Section 5 explains how the result extends to full linear logic with algebraic rules for weakening and contraction.

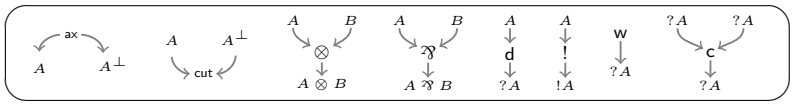


Figure 2 MELL links.

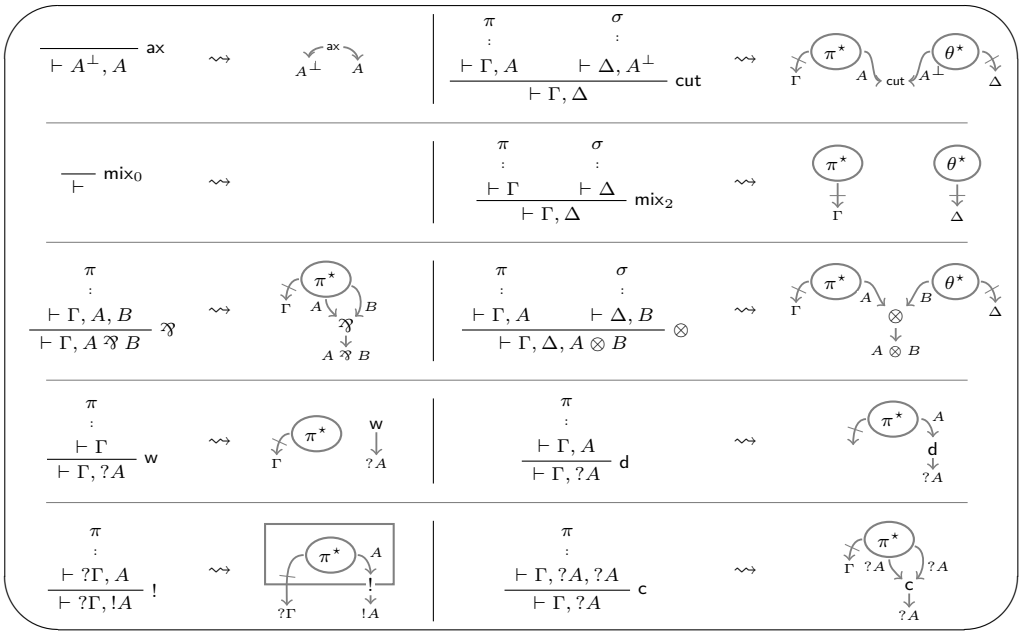


Figure 3 The translation of MELL sequent proofs to nets.

2 MELL Proof Nets

MELL. Multiplicative and Exponential Linear Logic (MELL) formulas are given by:

$$A, B, C \quad ::= \quad X \quad | \quad X^{\perp} \quad | \quad A \otimes B \quad | \quad A \parr B \quad | \quad !A \quad | \quad ?A$$

where X and $X^{\perp}$ are atomic formulas. For the sake of conciseness, the multiplicative units are not considered here, because their cut-elimination is trivial. The sequents of MELL are monolateral, *i.e.* they have the shape $\vdash \Gamma$, where Γ is a multiset of formulas. The rules of MELL are in Figure 3 (left side of every $\rightsquigarrow$); note the presence of the binary and nullary mix rules.

Nets. Nets are labelled directed graphs with *pending edges*, *i.e.* some edges may not have a source or a target, but not both. Nodes, called **links**, represent deductive rules and are labeled with an element of $\{\mathsf{ax}, \mathsf{cut}, 1, \bot, \otimes, \parr, !, \mathsf{d}, \mathsf{w}, \mathsf{c}\}$. Edges are labeled with a MELL formula. The label of a link forces the number and the labels of its incoming/outcoming edges as shown in Figure 2. The **conclusions (resp. premises)** of a link are those represented below (resp. above) the link symbol, *e.g.* the $\otimes$-link has two premises (labeled A and B), and one conclusion ($A \otimes B$). A **?-formula** is a formula $?A$ for some A, and a **?-link** is a link with an edge labeled with a ?-formula.

▶ **Definition 1** (MELL net with boxes)**.** A **(MELL) net** P is a finite set of links from those in Fig. 2 s.t. every edge is the conclusion of some link. The conclusion edges of P are its pending edges and the **conclusion links** of P are its links with pending edges.

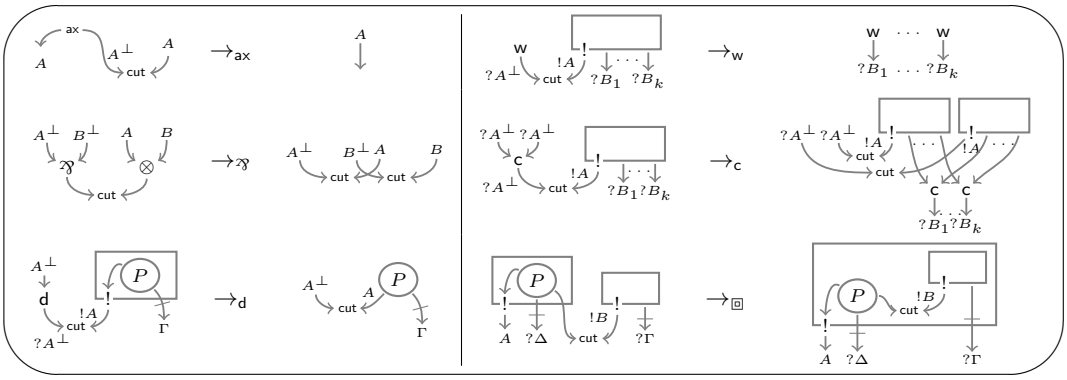


Figure 4 Black-box rewriting rules for MELL proof nets.

A **net with boxes** P is a net plus for any !-link l a subset $box(l)$ of the links of P, called the **box** of l, s.t. $l \in box(l)$ and:

- **Subnet**: its interior $int(l) := box(l) \setminus \{l\}$ is a net with boxes, whose boxes are inherited from P.
- **Border**: the premise of l is a conclusion of $int(l)$, and all other conclusions of $int(l)$ are labeled with ?-formulas;
- **Nesting**: For any two !-links l and i if $box(l) \cap box(i) \neq \emptyset$ then $box(l) \subseteq box(i)$ or $box(i) \subseteq box(l)$.

The translation from MELL proofs to nets with boxes is in Fig. 3, where the bar on some edges denotes a (multi)set of conclusions. The original version of the cut-elimination rules for MELL is in Figure 4. The rule $\rightarrow_{\boxdot}$ is the commutative rule. The union of all these rules is noted $\rightarrow_{com}$ (for containing the commutative rule).

The **level** of a link is the number of boxes in which it is contained (a !-link is not contained in its own box), and the level of a net P is the maximum level of a link of P. The **box address** of a link is the (possibly empty) sequence of !-boxes in which it is contained, starting from the outside. Note that boxes can be represented as labels if every link is decorated with its box address.

Proof nets. A net with boxes P is a **proof net** if it is the translation of a sequent calculus proof π. The proof of strong normalization will be by induction on an inductive decomposition of P, provided by π, and it will use the fact that proof nets reduce to proof nets. One of the motivations for considering MELL extended with the mix rules is that in this case there exist correctness criteria (easy adaptations of those in [11, 4]), while without them no criteria are known. The existence of a correctness criterion implies that the result can be made independent from sequent calculus (and that proof nets are stable by reduction). However, we will not discuss this issue any further.

The box-crossing principle. A crucial point of our approach is the removal of the black-box principle explained in the introduction. Note that we first defined nets, and only afterwards nets with boxes. For us, boxes are a sort of additional layer, a decoration. In the traditional presentation boxes have an *auxiliary port* for every ?-conclusion of their interior, which blocks the interaction between the inner and the outer world. In our presentation there are no auxiliary ports, and boxes do not interfere with links, in particular they shall not prevent links to interact through box borders. This box-crossing principle induces a new set of exponential cut-elimination rules, shown in Figure 5. Note that there is no commutative rule,

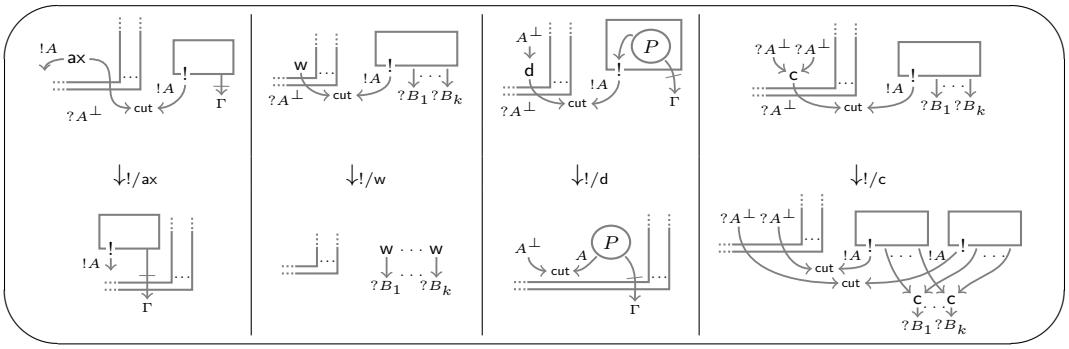


Figure 5 Box-crossing exponential rules for MELL proof nets.

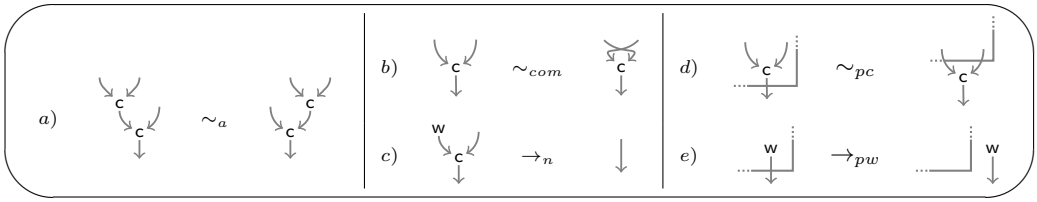


Figure 6 a–c) Co-associativity, co-commutativity, and co-neutrality for contractions; d–e) Permutations with box borders.

whose task is accomplished by the new axiom, dereliction, and weakening rules. Essentially, commutations are delayed as much as possible and then performed in a single big step together with the axiom/weakening/dereliction cut-elimination. Note that the axiom rule in Figure 4 provides an additional exponential case, when the cut axiom formula is $!A$ for some A. The union of $\to_{\mathsf{ax}}, \to_{\parr}, \to_{!/\mathsf{ax}}, \to_{!/\mathsf{w}}, \to_{!/\mathsf{d}}, \to_{!/\mathsf{c}}$ is denoted by $\to_{key}$ (for having only key—or principal—cut-elimination cases). This dynamics arises naturally when boxes are represented implicitly using *jumps* [2, 1], as in these cases the border of the box is not represented explicitly.

These two sets of rules are complemented by a set of equations and by two additional rules for contraction and weakening, presented in Figure 6. The equation and the rule in Figure 6.*d-e* permutes the structural rules with box borders, and will have an important role in the proof of strong normalization. The relations in Figure 6.*a-c*, instead, make contraction and weakening the operations of a co-commutative co-monoid on every type $?A$, *i.e.* they express co-associativity and co-commutativity of contractions and co-neutrality of weakening with respect to contraction. Beyond being algebraically natural and also semantically sound, these complementary operations are necessary in order simulate the synthetic ?-links of [32, 8] and obtain proper representations of λ-calculi or systems of explicit substitutions [10]. Lifting strong normalization to these enriched representations is usually tricky. In [9] Di Cosmo and Guerrini employ non-trivial arguments based on the so-called *geometry of interaction*. In [29] Pagani and Tranquilli have to use sophisticated techniques for rewriting modulo equivalence relations. In our case these additional equations and rules are necessary (see next section) but they will not require any heavy machinery.

We use $\equiv$ for the equivalence generated by the equations in Figure 6.*a*,*b*,*d*; the union of the rules in Figure 6.*c*,*e* considered modulo $\equiv$ is denoted by $\Rrightarrow$ (*i.e.* $\Rrightarrow := \equiv (\to_n \cup \to_{pw}) \equiv$); we use $\Rrightarrow_{key}$ for the union of $\to_{key}$, $\to_n$, and $\to_{pw}$ modulo $\equiv$ (*i.e.* $\Rrightarrow_{key} := \equiv (\to_{key} \cup \to_n \cup \to_{pw}) \equiv$).

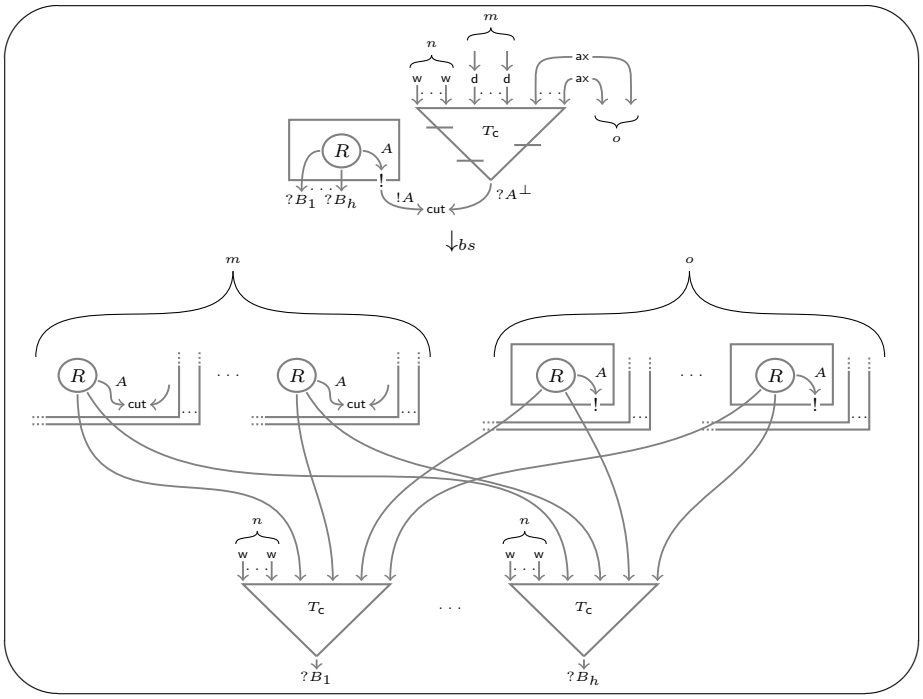


Figure 7 The big-step exponential cut-elimination rule.

3 Implicit Substitution and its Properties

In the literature the exponential cut-elimination rules sometimes appear in a big-step variant, where a box interacts with a whole tree of ?-links in just one shot. This rule was first designed to match substitution in λ-calculus [32, 8], and will be a crucial ingredient of our proof. The rule is usually presented collapsing whole trees of ?-formulas in just one node. The additional rules in Figure 6 are an alternative way of realizing such a collapse. We prefer them because the collapsed syntax is slightly *ad-hoc* with respect to box borders.

?-*trees.* Given an edge e of type $?A$ there is a unique maximal tree of ?-links rooted in e (this fact is a consequence of being a proof net, and would be easily seen if we were considering correctness). The internal nodes of such ?-trees are c-links and the leaves are $\{\mathsf{ax}, \mathsf{w}, \mathsf{d}\}$-links. According to the *box-crossing principle* these ?-trees may cross box borders. Graphically, this is represented by some horizontal lines crossing the tree, as in the triangle over the cut in the lhs of the rule in Figure 7 (the figure simplifies sightly the shape by grouping the leaves as axioms, weakenings, and derelictions, which is not necessarily the case). If the tree does not cross any box then the horizontal lines are omitted (as in the rhs of Figure 7). The intuition is that a ?-tree is the graphical analogous of a variable in a λ-term, whose occurrences are the leaves of the tree.

Big-step exponential rule. Figure 7 shows the big-step exponential rule. The idea is that the rule commutes the box and the ?-tree (as it is *natural* from a categorical point of view, see [24]) replacing every axiom leaf with a copy of the box, every dereliction leaf with a copy of the *interior* of the box, contracting the conclusions of the copies with copies of the cut ?-tree (now not crossing any box), and adding to these new trees a weakening for every weakening leaf in the old tree. Namely (an example is shown in Figure 8):

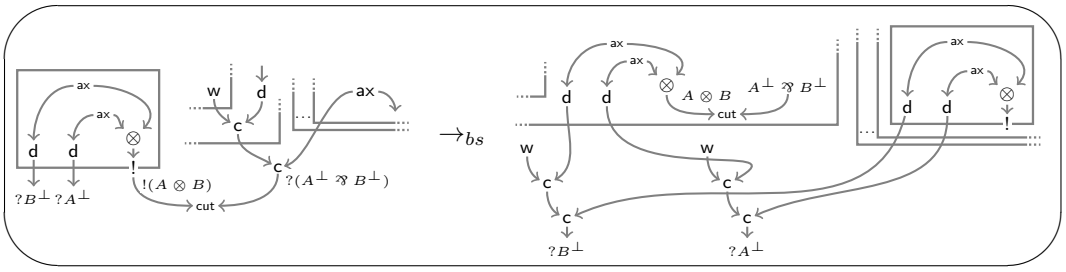


Figure 8 Example of substitution.

1. *Derelictions*: for $r \in \{1, \dots, m\}$ every dereliction l_r of the ?-tree is replaced by a copy R^r of R which is cut with the premise of l_r;
2. *Axioms*: for $s \in \{1, \dots, o\}$ every axiom i_s of the ?-tree is replaced by a copy B^s of the cut box;
3. *Contractions and weakenings*: for $q \in \{1, \dots, h\}$ the q-th conclusion of every R^r and that one of every B^s are contracted together via a copy of the cut ?-tree T_{c}, and the other n leaves of these copies of T_{c} are weakenings.
4. *Box addresses*: the replacements gives to R^r (resp. B^s) the exact box address of l_r (resp. i_s), and each copy of the ?-tree T_{c} and every copied weakening are put out of all the boxes crossed by T_{c}.

The reader who finds too vague the definition of substitution may prefer to think of boxes as labels on links indicating the box address: then the rule commutes the box and the ?-tree replacing for each copy (resp. opened copy) of the box the address prefix corresponding to the cut box with the address of the corresponding leaf of the ?-tree, and copies weakenings and contractions in T_x giving them the address of the cut box.

Explicit/implicit box substitution. To employ a handy notation for substitutions, we sometimes associate variable names to some ?-conclusions of a proof net, and denote with T_x a ?-tree on a conclusion x. Let then P be a proof net of conclusions $\vdash x : ?A, \Gamma$, and Q be a !-box of conclusions $\vdash ?B_1, \dots, ?B_n, !A^{\perp}$ of interior R. The **explicit subtitution** of Q to x in P, noted $P[x/Q]$ is obtained by simply cutting P and Q on x, as in the lhs of Figure 4.b. The **(implicit) substitution** $P\{x/Q\}$, is instead given by the rhs of the same figure. Because of the rule and the equation which bring contractions and weakenings out of boxes, we have to extend the definition of explicit and implicit substitutions to **extended boxes**, *i.e.* to the case where the proof net Q is a box B plus some ?-trees eventually having the auxiliary conclusions of B as leaves. In such a case the implicit (resp. explicit) substitution is simply obtained by substituting the box (resp. cutting the box) and adding to the result the same additional ?-trees.

Properties. Now, we study the properties of substitution that will be used in the proof of strong normalization. *None of the proofs in this section relies on types.*

The first property is a sort of commutation between promotion and substitution, which is assured by the rules permuting weakening and contractions with box borders. It is simple but important, and it is peculiar of proof nets as it concerns the structure of ?-trees. We need some definitions.

A **pointed net** is a net with a distinguished conclusion and $nets_A$ is the set of nets pointed on a conclusion of type A. We denote with $nets^?_A$ the subset of $nets_A$ composed by the proof nets which can be the interior of a box, *i.e.* whose non-distinguished conclusions are ?-formulas. For $P \in nets^?_A$ we write $!P$ for the proof net obtained by boxing P on A.

▶ **Lemma 2** (commutation). *Let $P \in nets^{?}_{A}$, x one of its auxiliary conclusions, and Q a boxed proof net. Then, $!(P\{x/Q\}) \Rrightarrow^{*} (!P)\{x/Q\}$.*

Proof. It is enough to push out of the outer box the contractions and weakenings of the copies of the ?-tree produced by substitution (if any). ◀

The second property is full composition, *i.e.* the fact that explicit substitutions can be executed and turned into implicit substitutions.

▶ **Lemma 3** (full composition). *Let P be a proof net and Q an extended box so that $P[x/Q]$ is well-defined. Then, $P[x/Q] \to^{*}_{key} P\{x/Q\}$, and so $P[x/Q] \Rrightarrow^{*}_{key} P\{x/Q\}$.*

Proof. The proof is by induction on the number of links in the ?-tree T_x on x (it goes essentially as Lemma 3.1, page 7, in [3]), which is also the exact number of steps in the reduction. If T_x is simply a leaf, *i.e.* a $\{\mathsf{ax}, \mathsf{d}, \mathsf{w}\}$-link, then the one-step reduct of the explicit substitution is exactly the implicit substitution. Otherwise, the extended box is cut with a contraction and the reduct has two cuts on smaller trees. The *i.h.* concludes the proof. ◀

The third and last property is the IE property. It requires a fine analysis of the commutation between $\Rrightarrow_{key}$ and the implicit substitution $P\{x/Q\}$, expressed by the following lemma. Essentially, it states that if $P \Rrightarrow_{key} P'$ then $P\{x/Q\} \Rrightarrow_{key} P'\{x/Q\}$ (and similarly for Q), but it points out some special cases and some additional information. Its only use is in the proof of the IE property, which follows.

▶ **Lemma 4** (Substitutivity). *Consider a proof net P and an extended box Q so that $P\{x/Q\}$ is defined.*

1. *$P \equiv P'$ implies $P\{x/Q\} \equiv P'\{x/Q\}$, and $Q \equiv Q'$ implies $P\{x/Q\} \equiv P\{x/Q'\}$.*
2. *$P \to_{key} P'$ implies $P\{x/Q\} \to_{key} \Rrightarrow^{*} P'\{x/Q\}$*
3. *Let $P \to_{n} P'$. If the step acts on T_x and Q has no auxiliary conclusion then $P\{x/Q\} = P'\{x/Q\}$, otherwise $P\{x/Q\} \to^{+}_{n} P'\{x/Q\}$.*
4. *Let $P \to_{pw} P'$. If the step acts on T_x then $P\{x/Q\} = P'\{x/Q\}$, otherwise $P\{x/Q\} \to_{pw} P'\{x/Q\}$.*
5. *Let $Q \Rrightarrow_{key} Q'$. If the step is strictly contained in the box of Q and every leaf of T_x is a weakening then $P\{x/Q\} \equiv P\{x/Q'\}$, otherwise $P\{x/Q\} \Rrightarrow^{+}_{key} P\{x/Q'\}$.*

Proof. 1. It is enough to prove the statement for the generators of $\equiv$—whose proof is given by easy verifications—as the result then follows by a straightforward induction.
2-4. The cut c reduced in $P \to_{key} P'$ clearly exists in $P\{x/Q\}$. For each point the only interesting case is when the reduction of c affects T_x, otherwise substitution and reduction simply commute. 2) A case analysis shows that if $P\{x/Q\} \to_{key} R$ by reducing the cut corresponding to c then R is equal to $P'\{x/Q\}$ up to the the rules and equations in Figure 6; 3) It is enough to repeat the $\to_n$-step on every copy of T_x in $P\{x/Q\}$; if Q has no auxiliary conclusion then the implicit substitution does not copy T_x, that explains the equality; 4) Substitution pushes weakenings and contractions out of boxes, so $P\{x/Q\} = P'\{x/Q\}$; if instead the step does not act on T_x then it commutes with substitution.
5. We show the property using only the rewriting rules, the statement then follows by point 1. Any reduction inside the box of Q has to be repeated for the $m + o$ copies of the interior of the cut box in $P\{x/Q\}$; this means 0 times if T_x has only weakening leaves. In the case of a $\to_{pw}$ step on the border of the box of Q, it is enough to do some $\to_{pw}$ steps and then some $\to_n$ steps, *i.e.* $P\{x/Q\} \to^{*}_{pw} \to^{*}_{n} P\{x/Q'\}$. The only remaining case is of a $\to_n$ step on one of the ?-trees extending the box of Q: the step simply commutes with substitution. ◀

We can now prove the IE property for $\to_{key}$. The contraction cut-elimination rule forces to prove a generalized n-ary formulation with respect to the one presented in the introduction.

Notations. SN_{key} denotes the set of proof nets which are strongly normalizing with respect to $\Rrightarrow_{key}$. For $R \in SN_{key}$ let $\eta(R)$ be the sum of the lengths of all $\Rrightarrow_{key}$-reductions from R, for $i \leq j$ let $\{\cdot\}_i^j := \{x_i/Q_i\}\dots\{x_j/Q_j\}$ and $[\cdot]_i^j := [x_i/Q_i]\dots[x_j/Q_j]$, and let T_{x_i} be the ?-tree on x_i for $i \in \{1,\dots,n\}$. The **size** $|T|$ of a ?-tree T is the number of $\{\mathsf{ax},\mathsf{w},\mathsf{d},\mathsf{c}\}$-links in T plus for every weakening l in T the number of boxes crossed by the path from l to the root of T (added to take into account rule $\to_{pw}$).

▶ **Lemma 5** (IE property). *Let P be a net of conclusions $\vdash \Gamma, x_1{:}?A_1, \dots, x_n{:}?A_n$ and $Q_1,\dots,Q_n \in SN_{key}$ be extended boxes of type $!A_1^\perp,\dots,!A_n^\perp$, respectively. Then $P\{\cdot\}_1^n \in SN_{key}$ implies $P[\cdot]_1^n \in SN_{key}$.*

Proof. The proof is by induction on the triple, lexicographically ordered (and it is similar to the one for Theorem 4.3, page 13, in [3]):

$$(\eta(P\{\cdot\}_1^n), \sum_{i=1}^n |T_{x_i}|, \sum_{i=1}^n \eta(Q_i))$$

The proof consists in showing that whenever $P[\cdot]_1^n \Rrightarrow_{key} R$ then $R \in SN_{key}$. For the reduction cases the measure decreases (so that we can apply the *i.h.*) and for the equivalence cases the measure is invariant, and so it properly lifts to equivalence classes. Cases:

1. *Equivalence in P*: $P[\cdot]_1^n \equiv P'[\cdot]_1^n$ because $P \equiv P'$. Then Lemma 4.1 gives $P\{\cdot\}_1^n \equiv P'\{\cdot\}_1^n$, and so the first component of the measure does not change. The second cannot be altered by $\equiv$ and the third one is not affected.
2. *Equivalence in Q_i*: analogous to the previous case.
3. *$\to_{key}$-reduction in P*: if $P[\cdot]_1^n \to_{key} P'[\cdot]_1^n$ then Lemma 4.2 gives $P\{\cdot\}_1^n \to_{key} \Rrightarrow_{key}^* P'\{\cdot\}_1^n$ and so $\eta(P'\{\cdot\}_1^n) < \eta(P\{\cdot\}_1^n)$. Then, the *i.h.* allows to conclude with $P'[\cdot]_1^n \in SN_{key}$.
4. *$\to_n$-reduction in P*: if the step acts on T_{x_i} (for some i) and Q_i has no auxiliary conclusions then by Lemma 4.3 $P\{\cdot\}_1^n = P'\{\cdot\}_1^n$, and so the first component of the measure does not change, but the size of T_{x_i} strictly decreases and so we conclude by the *i.h.*. Otherwise, Lemma 4.3 gives $P\{\cdot\}_1^n \to_n^+ P'\{\cdot\}_1^n$, and the first component decreases.
5. *$\to_{pw}$-reduction in P*: If the step acts on T_{x_i} for some i then by Lemma 4.4 $P\{\cdot\}_1^n = P'\{\cdot\}_1^n$, but the second component decreases. Otherwise, Lemma 4.4 gives $P\{\cdot\}_1^n \to_{pw} P'\{\cdot\}_1^n$ and the first component decreases.
6. *Reduction in Q_i*: by Lemma 4.5 there are two sub-cases. If every leaf of T_{x_i} is a weakening and the step is contained in the box of Q_i then the lemma gives $P\{\cdot\}_1^{i-1}\{x_i/Q_i\}\{\cdot\}_{i+1}^n = P\{\cdot\}_1^{i-1}\{x_i/Q_i'\}\{\cdot\}_{i+1}^n$, because the implicit substitution erases both Q_i and Q_i'. The second component of the measure does not change either. However, the third component decreases, because $\eta(Q_i') < \eta(Q_i)$, and we conclude using the *i.h.*. Otherwise, $P\{\cdot\}_1^{i-1}\{x_i/Q_i\}\{\cdot\}_{i+1}^n \to_{key}^+ P\{\cdot\}_1^{i-1}\{x_i/Q_i'\}\{\cdot\}_{i+1}^n$ and the first component decreases.
7. *Reduction of $[x_i/Q_i]$*: there are two sub-cases:
   a. *Cut with a contraction*: $P[\cdot]_1^{i-1}[x_i/Q_i][\cdot]_{i+1}^n \to_{key} P'[\cdot]_1^{i-1}[x_i'/Q_i][x_i''/Q_i][\cdot]_{i+1}^n$, where P' is the net obtained from P by removing the contraction on x_i, and x_i' and x_i'' are the names of the obtained two ?-trees. We can apply the *i.h.*, because $P'\{\cdot\}_1^{i-1}\{x_i'/Q_i\}\{x_i''/Q_i\}\{\cdot\}_{i+1}^n = P\{\cdot\}_1^n$ and the second component decreases, since $|T_{x_i'}| + |T_{x_i''}| < |T_{x_i}|$. Note that the third element of the measure potentially increases, so that the lexicographic order on the measure is necessary (in the previous case of a reduction in P, similarly, the second element may increase).

b. *Cut with an axiom/dereliction/weakening*: the reduct and the implicit substitution coincide, *i.e.* $P[\cdot]_1^{i-1}[x_i/Q_i][\cdot]_{i+1}^n \to_{key} P[\cdot]_1^{i-1}\{x_i/Q_i\}[\cdot]_{i+1}^n = P\{x_i/Q_i\}[\cdot]_1^{i-1}[\cdot]_{i+1}^n$. As in the previous sub-case the first component of the measure does not change, while the second decreases (one ?-tree less, and they always have positive size), and so we conclude by the *i.h.*. ◀

The IE property also holds for $\to_{com}$, because it is known that the system is SN, but the simple proof technique presented here for $\Rightarrow_{key}$ cannot be applied. The problem is that the box commutation rule breaks the inductive invariant. In fact, in the reduction of $[x_i/Q_i]$ one has to consider the case of a commutative step, which would bring Q_i inside P, getting a proof net P' for which $\eta(P'\{\cdot\}_1^{i-1}\{\cdot\}_{i+1}^n)$ is bigger than $\eta(P\{\cdot\}_1^n)$. Then for $\to_{com}$ a much more involved proof strategy has to be employed, like the labeling technique in [21].

4 Strong Normalization via Reducibility Candidates

Reducibility candidates are a standard construction, that we use following the schema in the literature [14, 13]. What is original here is the proof that every proof net is reducible. The method requires many definitions and notations.

A rewriting relation $\to$ for proof nets is **substitutive** if 1) multiplicative cuts are reduced only according to the usual cut-elimination rules; 2) it is defined for all cuts; 3) it makes promotion commute with the implicit substitution, *i.e.* $!(P\{x/Q\}) \to^* (!P)\{x/Q\}$; 4) it satisfies *full composition* and the *IE property*; 5) reduction is stable by subnets and by context closure. In the following $\to$ denotes a substitutive rewriting relation and every notion is parametrized by $\to$. However, to simplify the terminology and the notation *we keep the parametrization implicit as much as possible.*

Notations. If $P \in nets_A$ and $Q \in nets_{A^\perp}$ then $cut(P|Q)$ is the net obtained by cutting P and Q on their distinguished conclusions. We use $SN_\to$ for the set of $\to$-strongly normalizing proof nets and we define $SN_A := nets_A \cap SN_\to$. Moreover, $\eta(P)$ here denotes the sum of the length of all $\to$-reductions from P, for $P \in SN_\to$.

Duality. Given $S \subseteq nets_A$ the *dual* set $S^\perp \subseteq nets_{A^\perp}$ contains the nets Q s.t. $cut(Q|P) \in SN_\to$ for every $P \in S$. *Properties of duality*: if $S \neq \emptyset$ then $S^\perp \subseteq SN_\to$; $S \subseteq S^{\perp\perp}$; $S^{\perp\perp\perp} = S^\perp$; if $S \subseteq R$ then $R^\perp \subseteq S^\perp$.

▶ **Lemma 6** (non-emptyness). *If $S \subseteq SN_A$ then $S^\perp$ contains the axiom on $A^\perp$ (and A). Consequently, $S^\perp \neq \emptyset$.*

Proof. Let Q be the axiom on $A^\perp$ (and A). We show that $cut(P|Q) \in SN_\to$ for all $P \in S$; it then follows that $Q \in S^\perp$. By induction on $\eta(P)$, showing that any reduct of $cut(P|Q)$ is in $SN_\to$. Two cases. 1) *Reduction of the introduced cut*: we get $cut(P|Q) \to P$ and we conclude, since $P \in SN_\to$ by hypothesis. 2) *Reduction of a cut of P*: then $cut(P|Q) \to cut(P'|Q)$ and we conclude by the *i.h.*. ◀

A **reducibility candidate** is a set of pointed nets $S \subseteq nets_A$ s.t. $S = S^{\perp\perp}$, $S \neq \emptyset$ and $S \subseteq SN_A$. The general property $S^{\perp\perp\perp} = S^\perp$ provides the typical way of building reducibility candidates, consisting in taking the dual of a non-empty set of strongly normalizing nets.

We associate to every type A a reducibility candidate $[\![A]\!] \subseteq nets_A$, by induction on A:

- *Atomic formula*: $[\![X]\!] := SN_X$.
- *Tensor*: $[\![A \otimes B]\!] := \{P \otimes Q \mid P \in [\![A]\!], Q \in [\![B]\!]\}^{\perp\perp}$, where $P \otimes Q$ is the net obtained by adding a tensor link on the distinguished conclusions of P and Q.
- *Par*: $[\![A \parr B]\!] := [\![A^\perp \otimes B^\perp]\!]^\perp$.

- *Bang*: $[\![!A]\!] := \{!P \mid P \in [\![A]\!] \cap nets^{?}_{A}\}^{\perp\perp}$ (these notations are defined in the paragraph before Lemma 2).
- *Why not*: $[\![?A]\!] := [\![!A^{\perp}]\!]^{\perp}$.

▶ **Lemma 7.** *$[\![A]\!]$ is a reducibility candidate for every MELL formula A.*

Proof. By induction on the definition of $[\![A]\!]$. The base case follows from the more general fact that for whatever A, SN_A is the reducibility candidate given by $\{P\}^{\perp}$, where P is the axiom on $A^{\perp}$. The inductive cases follow easily by the *i.h.* and the properties of duality. ◀

More notations. Given a multiset of formulas $\Gamma = A_1, \dots, A_k$ we use Q^{Γ} for a multiset of pointed proof nets $Q_1, \dots, Q_k$ s.t. $Q_i \in nets_{A_i}$ for $i \in \{1, \dots, k\}$, and we use $Q^{\Gamma} \in [\![\Gamma]\!]$ if moreover $Q_i \in [\![A_i]\!]$. If P has conclusions Γ we also write $cut(P|Q_1, \dots, Q_k)$ or $cut(P|Q^{\Gamma})$ for the net obtained by cutting the conclusion A_i of P with Q_i for every $i \in \{1, \dots, k\}$.

A proof net P of conclusions $\vdash \Gamma$ is **(→-)reducible** if $cut(P|Q^{\Gamma}) \in SN_{\to}$ for every $Q^{\Gamma} \in [\![\Gamma]\!]$. Similarly, a proof net P pointed on A and of conclusions $\vdash A, \Gamma$ is reducible when $cut(P|Q^{\Gamma}) \in [\![A]\!]$ for every $Q^{\Gamma} \in [\![\Gamma]\!]$. *1st property of reducibility*: the two formulations of reducibility are related as follows. Let P be a proof net of conclusions $\vdash A, \Gamma$, and let P^{A} be P but pointed on A. Then, P is reducible iff P^{A} is reducible. *2nd property of reducibility*: Let P be a proof net of conclusions $\vdash ?A, \Gamma$. By the previous property P is reducible iff $P^{?A} \in [\![?A]\!]$. By definition, to show $P^{?A} \in [\![?A]\!]$ we have to cut P on $?A$ with a proof net in $\{!P \mid P \in [\![A]\!]\}$ (note the absence of the double dual), *i.e.* only with respect to nets contained in boxes. Similarly, for ⅋-formulas we only have to consider the case in which they are cut with a tensor.

The two properties of reducibility are repeatedly used in the following proof, and applied to many proof nets simultaneously. The novelties of the proof are the treatment of the exponential cases, in particular the case of !, and the parametrization with respect to a substitutive relation.

▶ **Theorem 8.** *Let $\to$ be a substitutive rewriting relation. Every proof net is $\to$-reducible.*

Proof. Let P be a proof net and π a sequent calculus proof mapping to P. The proof is by induction on π. The base cases:

- **Zeroary mix**: trivial, because the net is empty.
- **Axiom**: P is an axiom of conclusions $\vdash A, A^{\perp}$. Then we need to show that $cut(P|Q, R) \in SN_{\to}$ for $Q \in [\![A]\!]$ and $R \in [\![A^{\perp}]\!]$. By induction on $\eta(Q) + \eta(R)$, showing that whenever $cut(P|Q, R) \to S$ then $S \in SN_{\to}$. If reduction takes place in Q or in R then we conclude by the *i.h.*. Otherwise it is one of the two introduced axiom cuts which is reduced. The axiom is at level 0, so it is rule $\to_{\mathsf{ax}}$ which is applied. In both cases the reduct is $cut(Q|R)$, which is in $SN_{\to}$ by the properties of reducibility candidates.

The inductive exponential cases. Suppose that the last rule of π is a:

- **Promotion**. Consider P as a net pointed on $!A$. It writes as $!Q$, for Q of conclusions $\vdash ?B_1, \dots, ?B_n, A$. We need to show that $cut(!Q|S, R_1, \dots, R_n) \in SN_{\to}$ where $S \in [\![?A^{\perp}]\!]$ and (by the second property of reducibility) R_i is a box of distinguished conclusion $!B_i^{\perp}$. Note that $cut(!Q|S, R_1, \dots, R_n) = cut(!Q|S)[x_1/R_1] \dots [x_n/R_n]$ (shorted $cut(!Q|S)[\cdot]^n_1$), if $?B_i$ is named x_i for $i \in \{1, \dots, n\}$. Now, by *i.h.* Q is reductible, which implies $Q[\cdot]^n_1 \in [\![A]\!]$ and so $!(Q[\cdot]^n_1) \in [\![!A]\!]$. Note that in particular $cut(!(Q[\cdot]^n_1)|S) \in SN_{\to}$ holds. By full composition $cut(!(Q[\cdot]^n_1)|S) \to^* cut(!(Q\{\cdot\}^n_1)|S)$ and by commutation $cut(!(Q\{\cdot\}^n_1)|S) \to^* cut((!Q)\{\cdot\}^n_1|S) = cut(!Q|S)\{\cdot\}^n_1$, and so this last proof net is in $SN_{\to}$. For $i \in \{1, \dots, n\}$ we have $R_i \in SN_{\to}$, and so we can apply the IE property and get $(cut(!Q|S))[\cdot]^n_1 \in SN_{\to}$, which concludes the proof.

- **Weakening**, *i.e.* P is given by a proof net Q of conclusions $\vdash \Gamma$ plus a weakening l of conclusion $?A$, that we name x. By *i.h.* $cut(Q|R^\Gamma) \in SN_\to$ for every $R^\Gamma \in [\![\Gamma]\!]$. We need to show that $cut(P|R^\Gamma) \in [\![?A]\!]$, *i.e.* if we name x the conclusion of the weakening we need to show that $cut(P|R^\Gamma)[x/!S] \in SN_\to$ for $S \in [\![A^\perp]\!] \cap nets^?_{A^\perp}$ (remark the use of the 2nd property of reducibility). We get $cut(P|R^\Gamma)[x/!S] \to^* cut(P|R^\Gamma)\{x/!S\}$ by full composition, where $cut(P|R^\Gamma)\{x/!S\}$ is $cut(Q|R^\Gamma)$ plus a weakening at level 0 for every non-pointed conclusion of S. The proof net $cut(Q|R^\Gamma)$ is in SN_{key} by *i.h.*, which implies $cut(P|R^\Gamma)\{x/!S\} \in SN_\to$. We also know that S, and thus $!S$, is in $SN_\to$. The IE property then gives $cut(P|R^\Gamma)[x/!S] \in SN_\to$, *i.e.* $cut(P|R^\Gamma) \in [\![?A]\!]$.
- **Dereliction**, *i.e.* P is given by a proof net Q of conclusions $\vdash A, \Gamma$ plus a dereliction on A, whose conclusion $?A$ we name x. We need to show that $cut(P|R^\Gamma) \in [\![?A]\!]$, *i.e.* that $cut(P|R^\Gamma)[x/!S] \in SN_\to$ for $S \in [\![A]\!] \cap nets^?_{A^\perp}$. By full composition $cut(P|R^\Gamma)[x/!S] \to^* cut(P|R^\Gamma)\{x/!S\} = cut(Q|R^\Gamma, S)$. By *i.h.* this last net is in $SN_\to$ and by hypothesis $S \in SN_\to$, so we can apply the IE property and get $cut(P|R^\Gamma)[x/!S] \in SN_\to$.
- **Contraction**, *i.e.* P is given by a proof net Q of conclusions $\vdash ?A, ?A, \Gamma$ plus a contraction on the two occurrences of $?A$. Let us call these occorrences y and z. The *i.h.* is $cut(Q|R^\Gamma)[y/!R_1][z/!R_2] \in SN_\to$ for every $R_1, R_2 \in [\![A]\!] \cap nets^?_{A^\perp}$ and for every $R^\Gamma \in [\![\Gamma]\!]$. We need to show that $cut(P|R^\Gamma) \in [\![?A]\!]$, which is equivalent to $cut(P|R^\Gamma)[x/!S] \in SN_\to$ for every $S \in [\![A^\perp]\!] \cap nets^?_{A^\perp}$, if we name x the conclusion of the contraction. We get $cut(P|R^\Gamma)[x/!S] \to^* cut(P|R^\Gamma)\{x/!S\}$ by full composition. Note that $cut(P|R^\Gamma)\{x/!S\}$ is equal to $cut(Q|R^\Gamma)\{y/!S\}\{z/!S\}$ and that this net is in $SN_\to$, because the proof net $cut(Q|R^\Gamma)[y/!S][z/!S]$ (which is SN by *i.h.*) reduces to it by full composition. We also know that S, and thus $!S$, is in $SN_\to$. Then we can apply the IE property, getting $cut(P|R^\Gamma)[x/!S] \in SN_\to$.

We only show one non-exponential case (for $\otimes$) as the others ($\parr$, cut, and binary mix) are simpler and similar to what already appeared in the literature. If the last rule of π is a tensor then P writes as $Q_1 \otimes Q_2$. Let $\vdash \Gamma, \Delta, A \otimes B$, $\vdash \Gamma, A$, and $\vdash \Delta, B$ be the conclusions of P, Q_1, and Q_2, respectively. We need to show that $cut(Q_1 \otimes Q_2|R^\Gamma, R^\Delta) \in [\![A \otimes B]\!]$. But $cut(Q_1 \otimes Q_2|R^\Gamma, R^\Delta) = cut(Q_1|R^\Gamma) \otimes cut(Q_2|R^\Delta)$, and by *i.h.* $cut(Q_1|R^\Gamma) \in [\![A]\!]$ and $cut(Q_2|R^\Delta) \in [\![B]\!]$. Then by definition of $[\![A \otimes B]\!]$ we get $cut(Q_1 \otimes Q_2|R^\Gamma, R^\Delta) \in [\![A \otimes B]\!]$ ◀

▶ **Corollary 9.** *Every proof net is in $SN_\to$, and so in SN_{key}.*

Proof. Let P be a proof net. By the previous theorem P is reductible. If $\vdash A_1, \ldots, A_n$ are its conclusions then $P \in [\![A_i]\!]$ for $i \in \{1, \ldots, n\}$. By Lemma 6 $[\![A_i^\perp]\!]$ contains the axiom on $A_i^\perp$. Then cutting P with axioms on $A_1^\perp, \ldots, A_n^\perp$ we get a net Q s.t. $Q \in SN_\to$ and $Q \to^* P$. So, $P \in SN_\to$. Section 3 shows that $\to_{key}$ is substitutive, and so $P \in SN_{key}$. ◀

5 Variations on a Theme

The multiplicative units are in fact already inside MELL: 1 can be coded with the proof of $!(A \parr A^\perp)$ obtained by an axiom on A, and $\perp$ can be coded by a weakening on $?(A^\perp \otimes A)$.

Second order quantifiers. the extension of the proof to second order can be done as in [14], it is standard, not particularly interesting, and it obfuscates the structure of the proof by forcing to deal with substitution on type variables. Once the reducibility candidates technique is adopted the treatment of second order is smooth.

Additives. If they are represented using slices (as in [22], for instance) their treatment is trivial, because the additive cut-elimination rules strictly decrease the size of the proof net. Replacing slices by additive boxes may seem trickier, as it requires some commutation

rules. But in [28] it is shown that *SN* for the additive slices implies *SN* for the additive boxes (Proposition 5.6, page 56). Thus, in either cases we catch strong normalization for full linear logic. Our proof technique does not rely on confluence, which is why the extension is so simple. This is an impressive simplification of the involved proof by Tortora de Falco and Pagani in [28], because they use a conservation theorem, that requires to show a delicate form of confluence, and confluence is notoriously problematic with the additives.

Confluence. In the case without exponential axioms, confluence (and even Church-Rosser modulo $\equiv$) can be shown along the lines of the study in [1]. Exponential axioms introduce a new critical pair that is unusual and difficult to study. We claim that confluence still holds. Preliminary results suggest that the notion of implicit substitution can be used to obtain a proof of confluence by projection, following the usual argument for explicit substitutions.

6 Conclusions

We gave a presentation of MELL proof nets without any commutative rule, and showed a proof of strong normalization which is simple, informative, modular, and does not rely on confluence. The cut-elimination theorem for proof nets is now not just proved, but also understood and made accessible to a wider audience.

Acknowledgements

To Delia Kesner, who taught me the art of strong normalization proofs, and with whom I had uncountable discussions on commutative rules, normalization, and proof nets. And to the anonymous reviewers, who pointed out various inaccuracies. I also had fruitful discussions with Michele Pagani, Lorenzo Tortora de Falco, Paolo Tranquilli, Olivier Laurent, Stefano Guerrini, Stéphane Gimenez, and Frank Pfenning. This work was partially supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168.

References

1 Beniamino Accattoli. Compressing polarized boxes. Accepted at LICS, 2013.
2 Beniamino Accattoli and Stefano Guerrini. Jumping boxes. In *CSL*, pages 55–70, 2009.
3 Beniamino Accattoli and Delia Kesner. Preservation of strong normalisation modulo permutations for the structural λ-calculus. *Logical Methods in Computer Science*, 8(1), 2012.
4 Andrea Asperti. Causal dependencies in multiplicative linear logic with mix. *Mathematical Structures in Computer Science*, 5(3):351–380, 1995.
5 Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.
6 Eduardo Bonelli. Perpetuality in a named lambda calculus with explicit substitutions. *Mathematical Structures in Computer Science*, 11(1):47–90, 2001.
7 Vincent Danos. *La Logique Linéaire appliqué à l'étude de divers processus de normalisation (principalement du λ-calcul)*. Phd thesis, Université Paris 7, 1990.
8 Vincent Danos and Laurent Regnier. Proof-nets and the Hilbert space. In *Advances in Linear Logic*, pages 307–328. Cambridge University Press, 1995.
9 Roberto Di Cosmo and Stefano Guerrini. Strong normalization of proof nets modulo structural congruences. In *RTA*, pages 75–89, 1999.
10 Roberto Di Cosmo, Delia Kesner, and Emmanuel Polonovski. Proof nets and explicit substitutions. *Math. Str. in Comput. Sci.*, 13(3):409–450, 2003.
11 Arnaud Fleury and Christian Retoré. The mix rule. *Mathematical Structures in Computer Science*, 4(2):273–285, 1994.

12 R. O. Gandy. Proofs of strong normalization. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press Limited, 1980.

13 Stéphane Gimenez. Realizability proof for normalization of full differential linear logic. In *TLCA*, pages 107–122, 2011.

14 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

15 Jean-Yves Girard. *Proof theory and logical complexity*. Studies in proof theory. Bibliopolis, 1987.

16 Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.

17 Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

18 Dominic J. D. Hughes and Rob J. van Glabbeek. Proof nets for unit-free multiplicative-additive linear logic. *ACM Trans. Comput. Log.*, 6(4):784–842, 2005.

19 Jean-Baptiste Joinet. *Étude de la normalisation du calcul des séquents*. Phd thesis, Université Paris 7, 1977.

20 Delia Kesner. The theory of calculi with explicit substitutions revisited. In *CSL*, pages 238–252, 2007.

21 Delia Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3), 2009.

22 Olivier Laurent and Lorenzo Tortora de Falco. Slicing polarized additive normalization. In *Linear Logic in Computer Science*, volume 316 of *London Mathematical Society Lecture Note Series*, pages 247–282. Cambridge University Press, November 2004.

23 Ralph Loader. Normalisation by calculation. Unpublished note, available at `http://homepages.ihug.co.nz/~suckfish/papers/normal.pdf`, 1995.

24 Paul-André Melliès. Functorial boxes in string diagrams. In *CSL*, pages 1–30, 2006.

25 Robert Pieter Nederpelt. *Strong Normalization for a Typed Lambda Calculus with Lambda Structured Types*. Ph.D. thesis, Technische Hogeschool Eindhoven, 1973.

26 Mitsuhiro Okada. Phase semantic cut-elimination and normalization proofs of first- and higher-order linear logic. *Theor. Comput. Sci.*, 227(1-2):333–396, 1999.

27 Michele Pagani. The cut-elimination theorem for differential nets with promotion. In *TLCA*, pages 219–233, 2009.

28 Michele Pagani and Lorenzo Tortora de Falco. Strong normalization property for second order linear logic. *Theor. Comput. Sci.*, 411(2):410–444, 2010.

29 Michele Pagani and Paolo Tranquilli. The conservation theorem for differential nets. Accepted for publication in Math. Str. in Comput. Sci., 2009.

30 Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Carnegie Mellon University, 1994.

31 Frank Pfenning. Structural cut elimination. In *LICS*, pages 156–166, 1995.

32 Laurent Regnier. *Lambda-calcul et réseaux*. PhD thesis, Univ. Paris VII, 1992.

33 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

34 D. T. van Daalen. *The language theory of automath*. Phd thesis, Technische Hogeschool Eindhoven, 1977.

35 Femke van Raamsdonk. *Confluence and Normalization for Higher-Order Rewriting*. PhD thesis, Amsterdam Univ., Netherlands, 1996.

36 Femke van Raamsdonk and Paula Severi. On normalisation. Technical Report CS-R9545, Centrum Wiskunde & Informatica, 1995.

37 Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. Perpetual reductions in lambda-calculus. *Inf. Comput.*, 149(2):173–225, 1999.

A Combination Framework for Complexity*

Martin Avanzini and Georg Moser

Institute of Computer Science,
University of Innsbruck, Austria
{martin.avanzini,georg.moser}@uibk.ac.at

Abstract

In this paper we present a combination framework for the automated polynomial complexity analysis of term rewrite systems. The framework covers both *derivational* and *runtime complexity* analysis, and is employed as theoretical foundation in the automated complexity tool TcT. We present generalisations of powerful complexity techniques, notably a generalisation of *complexity pairs* and *(weak) dependency pairs*. Finally, we also present a novel technique, called *dependency graph decomposition*, that in the dependency pair setting greatly increases modularity.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases program analysis, term rewriting, complexity analysis, automation

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.55

1 Introduction

In order to measure the complexity of a term rewrite system (TRS for short) it is natural to look at the maximal length of derivation sequences—the *derivation length*—as suggested by Hofbauer and Lautemann in [15]. The resulting notion of complexity is called *derivational complexity*. Hirokawa and the second author introduced in [12] a variation, called *runtime complexity*, that only takes *basic* or *constructor-based* terms as start terms into account. The restriction to basic terms allows one to accurately express the complexity of a program through the runtime complexity of a TRS. Noteworthy both notions constitute an *invariant cost model* for rewrite systems [10, 4].

The body of research in the field of complexity analysis of rewrite systems provides a wide range of different techniques to analyse the time complexity of rewrite systems, fully automatically. Techniques range from *direct methods*, like *polynomial path orders* [3, 5] and other suitable restrictions of termination orders [9, 20], to *transformation techniques*, maybe most prominently adaptions of the *dependency pair method* [12, 14, 21], *semantic labeling* over finite carriers [2], methods to combine base techniques [24] and the *weight gap principle* [12, 24]. (See [19] for an overview of complexity analysis methods for term rewrite systems.) In particular the dependency pair method for complexity analysis allows for a wealth of techniques originally intended for termination analysis. We mention *(safe) reduction pairs* [12, 14], *various rule transformations* [21], and *usable rules* [12, 14]. Some very effective methods have been introduced specifically for complexity analysis in the context of dependency pairs. For instance, *path analysis* [12, 13, 14] decomposes the analysed rewrite relation into simpler ones, by treating paths through the *dependency graph* independently. *Knowledge propagation* [21] is another complexity technique relying on dependency graph

* This work was partially supported by FWF (Austrian Science Fund) project I-603-N18.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 55–70

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

analysis, which allows one to propagate bounds for specific rules along the dependency graph. Besides these, various minor simplifications are implemented in tools, mostly relying on dependency graph analysis. With this paper, we provide following contributions.

1. We propose a uniform *combination framework for complexity analysis*, that is capable of expressing the majority of the rewriting based complexity techniques in a unified way. Such a framework is essential for the development of a modern complexity analyser for term rewrite systems. The implementation of our complexity analyser TcT [7], the *Tyrolean Complexity Tool*, closely follows the formalisation proposed in this work. Noteworthy, TcT is currently the only tool that participates in all four complexity sub-divisions of the annual *termination competition*.[1]
2. A majority of the cited techniques were introduced in restricted or incompatible contexts. For instance, in [24] the derivational complexity of relative TRSs is considered. Conversely, neither [12, 14] nor [21] treat relative systems, and restrict their attention to basic start terms. Where non-obvious, we generalise these techniques to our setting. Noteworthy, our notion of *$\mathcal{P}$-monotone complexity pair* generalises complexity pairs from [24] for derivational complexity, *μ-monotone complexity pairs* for runtime complexity analysis [14], and *safe reduction pairs* studied in [12, 21] that work on dependency pairs.[2] We also generalise the two different forms of dependency pairs for complexity analysis introduced in [12] and [21]. This for instance allows our tool TcT to employ these powerful techniques on a TRS $\mathcal{R}$ relative to some theory expressed as a TRS $\mathcal{S}$.
3. We introduce a novel proof technique for runtime-complexity analysis called *dependency graph decomposition*. Resulting sub-problems are syntactically of a simpler form, and the analysis of these sub-problems is often easier. Importantly, the sub-problems are usually also computationally simpler in the sense that their complexity is strictly smaller than the one of the input problem. If the complexity of the two generated sub-problems is bounded by a function in $\mathsf{O}(f)$ and $\mathsf{O}(g)$ respectively, then the complexity of the input is bounded by $\mathsf{O}(f \cdot g)$. Experiments conducted with TcT indicate that this estimation is often asymptotically precise.[3]

This paper is structured as follows. In the next section we cover some basics. Our *combination framework* is then introduced in Section 3. In Section 4 we introduce *$\mathcal{P}$-monotone complexity pairs*. In Section 5 we introduce *dependency pairs for complexity analysis*, and reprove soundness of weak dependency pairs and dependency tuples. In Section 6 we introduce *dependency graph decomposition*, and conclude in Section 7.

Due to space limitations some proofs are only sketched, or have been completely omitted. The reader is kindly referred to the technical report [6], where proofs are given in full detail.

2 Preliminaries

Let R be a binary relation. The transitive closure of R is denoted by R^+ and its transitive and reflexive closure by R^*. For $n \in \mathbb{N}$ we denote by R^n the *n-fold composition of R*. The binary relation R is *well-founded* (on a set A) if there exists no infinite chain $a_0, a_1, \ldots$ with

[1] `http://www.termination-portal.org/wiki/Termination_Competition/`.
[2] In [21] safe reductions pairs are called COM-*monotone reduction pairs*.
[3] Detailed experimental evidence is provided online under `http://cl-informatik.uibk.ac.at/software/tct/experiments/tct2`.

$a_i \; R \; a_{i+1}$ for all $i \in \mathbb{N}$ ($a_0 \in A$). The relation R is *finitely branching* if for all elements a, the set $\{b \mid a \; R \; b\}$ is finite. A *preorder* is a reflexive and transitive binary relation.

We assume familiarity with rewriting [8] and just fix notations. We denote by $\mathcal{V}$ a countably infinite set of *variables* and by $\mathcal{F}$ a *signature*. The signature $\mathcal{F}$ and variables $\mathcal{V}$ are fixed throughout the paper, the set of terms over $\mathcal{F}$ and $\mathcal{V}$ is written as $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Throughout the following, we suppose a partitioning of $\mathcal{F}$ into *constructors* $\mathcal{C}$ and *defined symbols* $\mathcal{D}$. The set of *basic terms* $f(s_1, \dots, s_n)$, where $f \in \mathcal{D}$ and arguments s_i ($i = 1, \dots, n$) contain only variables or constructors, is denoted by $\mathcal{T}_\mathsf{b}$. Terms are denoted by $s, t, \dots$, possibly followed by subscripts. We use $s|_p$ to refer to the *subterm* of s at position p. We denote by $|t|$ the *size* of t, i.e., the number of occurrences of symbols in t. A *rewrite relation* $\to$ is a binary relation on terms closed under contexts and stable under substitutions. We use $\mathcal{R}, \mathcal{S}, \mathcal{Q}, \mathcal{W}$ to refer to *term rewrite systems* (*TRSs* for short). We denote by $\mathrm{NF}(\mathcal{R})$ the *normal forms* of $\mathcal{R}$, and abusing notation we extend this notion to binary relations $\to$ on terms in the obvious way. For a set of terms $T \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$, we define $\to(T) := \{t \mid \exists s \in T.\ s \to t\}$.

For two TRSs $\mathcal{Q}$ and $\mathcal{R}$, we define $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ if there exists a context C, substitution σ, and rule $f(l_1, \dots, l_n) \to r \in \mathcal{R}$ such that $s = C[f(l_1\sigma, \dots, l_n\sigma)]$, $t = C[r\sigma]$ and all arguments $l_i\sigma$ ($i = 1, \dots, n$) are $\mathcal{Q}$ normal forms. If $\mathcal{Q} = \varnothing$, we sometimes drop $\mathcal{Q}$ and write $\to_{\mathcal{R}}$ instead of $\xrightarrow{\varnothing}_{\mathcal{R}}$. Note that $\to_{\mathcal{R}}$ corresponds to the usual definition of rewrite relation of $\mathcal{R}$. The innermost rewrite relation of a TRS $\mathcal{R}$ is given by $\xrightarrow{\mathcal{R}}_{\mathcal{R}}$. We extend $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ to a relative setting and define for TRSs $\mathcal{R}$ and $\mathcal{S}$ the relation $\xrightarrow{\mathcal{Q}}_{\mathcal{R}/\mathcal{S}} := \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{S}} \cdot \xrightarrow{\mathcal{Q}}_{\mathcal{R}} \cdot \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{S}}$, and call $\xrightarrow{\mathcal{Q}}_{\mathcal{R}/\mathcal{S}}$ the *$\mathcal{Q}$-restricted rewrite relation of $\mathcal{R}$ modulo $\mathcal{S}$*.

To compare partial functions we use *Kleene equality*: two partial functions $f, g : \mathbb{N} \to \mathbb{N}$ are equal, in notation $f \simeq g$, if for all $n \in \mathbb{N}$ either $f(n)$ and $g(n)$ are defined and $f(n) = g(n)$, or both $f(n)$ and $g(n)$ are undefined. The *derivation height* of a term t with respect to a binary relation $\to$ on terms is given by $\mathsf{dh}(t, \to) \simeq \max\{n \mid \exists t_1, \dots, t_n.\ t \to t_1 \to \cdots \to t_n\}$. We emphasise that $\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{R}/\mathcal{S}})$, if defined, binds the number of $\mathcal{R}$ steps in all $\xrightarrow{\mathcal{Q}}_{\mathcal{R}\cup\mathcal{S}}$ derivations starting from t. We emphasise that our techniques always imply that $\mathsf{dh}(t, \to)$ is well-defined. Let T be a set of terms, and define $\mathsf{cp}(n, T, \to) := \max\{\mathsf{dh}(t, \to) \mid \exists t \in T,\ |t| \leqslant n\}$. The *derivational complexity* of a TRS $\mathcal{R}$ is given by $\mathrm{dc}_{\mathcal{R}}(n) := \mathsf{cp}(n, \mathcal{T}(\mathcal{F}, \mathcal{V}), \to_{\mathcal{R}})$ for all $n \in \mathbb{N}$, the *runtime complexity* takes only basic terms as starting terms T into account: $\mathrm{rc}_{\mathcal{R}}(n) := \mathsf{cp}(n, \mathcal{T}_\mathsf{b}, \to_{\mathcal{R}})$ for all $n \in \mathbb{N}$. By exchanging $\to_{\mathcal{R}}$ with $\xrightarrow{\mathcal{R}}_{\mathcal{R}}$ we obtain the notions of *innermost derivational* or *runtime complexity* respectively.

3 The Combination Framework

At the heart of our framework lies the notion of *complexity processor*, or simply *processor*. A complexity processor dictates how to transform the analysed input *problem* into sub-problems (if any), and how to relate the complexity of the obtained sub-problems to the complexity of the input problem. In our framework, such a processor is modeled as a set of inference rules

$$\frac{\vdash \mathcal{P}_1 \colon f_1 \quad \cdots \quad \vdash \mathcal{P}_n \colon f_n}{\vdash \mathcal{P} \colon f} \;,$$

over judgements of the form $\vdash \mathcal{P} \colon f$. Here $\mathcal{P}$ denotes a *complexity problem* (*problem* for short) and $f : \mathbb{N} \to \mathbb{N}$ a *bounding function*. The validity of a judgement $\vdash \mathcal{P} \colon f$ is given when the function f binds the complexity of the problem $\mathcal{P}$ asymptotically.

Conceptually, a complexity problem $\mathcal{P}$ consists of a set of *starting terms* $\mathcal{T}$ together with a relation $\xrightarrow{\mathcal{Q}}_{\mathcal{S}\cup\mathcal{W}}$ for TRSs $\mathcal{S}, \mathcal{W}$ and $\mathcal{Q}$. The complexity function $\mathsf{cp}_{\mathcal{P}} : \mathbb{N} \to \mathbb{N}$ of $\mathcal{P}$ accounts for the number of applications of rules from $\mathcal{S}$ in derivations starting from terms $t \in \mathcal{T}$, measured in the size of t.

▶ **Definition 3.1** (Complexity Problem, Complexity Function).

1. A *complexity problem* $\mathcal{P}$ (*problem* for short) is a quadruple $\langle \mathcal{S}, \mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$, in notation $\langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$, where $\mathcal{S}, \mathcal{W}, \mathcal{Q}$ are TRSs and $\mathcal{T} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$ a set of terms.
2. The *complexity (function)* $\mathsf{cp}_{\mathcal{P}} : \mathbb{N} \to \mathbb{N}$ of $\mathcal{P}$ is defined as the partial function

$$\mathsf{cp}_{\mathcal{P}}(n) := \mathsf{cp}(n, \mathcal{T}, \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}) \,.$$

In the sequel $\mathcal{P}$, possibly followed by subscripts, always denotes a complexity problem. Consider a problem $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$. We call $\mathcal{S}$ and $\mathcal{W}$ the *strict* and *weak component* of $\mathcal{P}$ respectively. The set $\mathcal{T}$ is called the set of *starting terms* of $\mathcal{P}$. We sometimes write $l \to r \in \mathcal{P}$ for $l \to r \in \mathcal{S} \cup \mathcal{W}$, and we denote by $\to_{\mathcal{P}}$ the rewrite relation $\xrightarrow{\mathcal{Q}}_{\mathcal{S} \cup \mathcal{W}}$. A derivation $t \to_{\mathcal{P}} t_1 \to_{\mathcal{P}} \cdots$ is also called a $\mathcal{P}$*-derivation* (*starting from* t). Observe that the derivational complexity of a TRS $\mathcal{R}$ corresponds to the complexity function of $\langle \mathcal{R}/\varnothing, \varnothing, \mathcal{T}(\mathcal{F}, \mathcal{V}) \rangle$. By exchanging the set of starting terms to basic terms we can express the *runtime complexity* of a TRS $\mathcal{R}$. If the starting terms are all basic terms, we call such a problem also a *runtime complexity problem*. Likewise, we can treat innermost rewriting by using $\mathcal{Q} = \mathcal{R}$. For the case $\mathrm{NF}(\mathcal{Q}) \subseteq \mathrm{NF}(\mathcal{S} \cup \mathcal{W})$, that is when $\to_{\mathcal{P}}$ is included in the innermost rewrite relation of $\mathcal{R} \cup \mathcal{S}$, we also call $\mathcal{P}$ an *innermost complexity problem*.

▶ **Example 3.2.** Consider the rewrite system $\mathcal{R}_{\times}$ given by the four rules

$$\mathsf{1}\colon 0 + y \to y \qquad \mathsf{2}\colon \mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad \mathsf{3}\colon 0 \times y \to 0 \qquad \mathsf{4}\colon \mathsf{s}(x) \times y \to (x \times y) + y \,,$$

and let $\mathcal{T}_{\mathsf{b}}$ denote basic terms with defined symbols $+, \times$ and constructors $\mathsf{s}, 0$. Then $\mathcal{P}_{\times} := \langle \mathcal{R}_{\times}/\varnothing, \mathcal{R}_{\times}, \mathcal{T}_{\mathsf{b}} \rangle$ is an innermost runtime complexity problem, in particular the complexity of $\mathcal{P}$ equals the innermost runtime complexity of $\mathcal{R}_{\times}$.

Note that even if $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ is terminating, the complexity function is not necessarily defined on all inputs. For a counter example, consider the problem $\mathcal{P}_1 := \langle \mathcal{S}_1/\mathcal{W}_1, \varnothing, \{\mathsf{f}(\bot)\} \rangle$ where $\mathcal{S}_1 := \{\mathsf{g}(\mathsf{s}(x)) \to \mathsf{g}(x)\}$ and $\mathcal{W}_1 := \{\mathsf{f}(x) \to \mathsf{f}(\mathsf{s}(x)),\ \mathsf{f}(x) \to \mathsf{g}(x)\}$. Note that for all $n \in \mathbb{N}$, maximal $\to_{\mathcal{P}_1}$ derivations are of the form

$$\mathsf{f}(\bot) \to^*_{\mathcal{W}_1} \mathsf{f}(\mathsf{s}^n(\bot)) \to_{\mathcal{W}_1} \mathsf{g}(\mathsf{s}^n(\bot)) \to^n_{\mathcal{S}_1} \mathsf{g}(\bot) \,.$$

Hence $\mathsf{f}(\bot) \to^n_{\mathcal{S}_1/\mathcal{W}_1} \mathsf{g}(\bot)$ holds for all $n \in \mathbb{N}$. Whereas $\to_{\mathcal{S}_1/\mathcal{W}_1}$ is well-founded, the above family of derivations shows that $\mathsf{cp}_{\mathcal{P}_1}(m) \simeq \mathsf{dh}(\mathsf{f}(\bot), \to_{\mathcal{S}_1/\mathcal{W}_1})$ is undefined for $m \geqslant 2$. If $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ is well-founded and *finitely branching* then $\mathsf{cp}_{\mathcal{P}}$ is defined on all inputs, by König's Lemma. This condition is sufficient but not necessary. The complexity function of the problem $\mathcal{P}_2 := \langle \mathcal{S}_2/\mathcal{W}_1, \varnothing, \{\mathsf{f}(\bot)\} \rangle$, where $\mathcal{S}_2 := \{\mathsf{g}(x) \to x\}$, is constant but $\mathsf{f}(\bot) \to_{\mathcal{S}_2/\mathcal{W}_1} \mathsf{s}^n(\bot)$ for all $n \in \mathbb{N}$, i.e, $\to_{\mathcal{S}_2/\mathcal{W}_1}$ is not finitely branching. In this work we do not presuppose that the complexity function is defined on all inputs, instead, this will be determined by our methods.

▶ **Definition 3.3** (Judgement, Processor, Proof).

1. A *(complexity) judgment* is a statement $\vdash \mathcal{P}\colon f$ where $\mathcal{P}$ is a complexity problem and $f : \mathbb{N} \to \mathbb{N}$. The judgment is *valid* if $\mathsf{cp}_{\mathcal{P}}$ is defined on all inputs, and $\mathsf{cp}_{\mathcal{P}} \in \mathsf{O}(f)$.
2. A *complexity processor* Proc (*processor* for short) is an inference rule

$$\frac{\vdash \mathcal{P}_1\colon f_1 \quad \cdots \quad \vdash \mathcal{P}_n\colon f_n}{\vdash \mathcal{P}\colon f}\ \mathsf{Proc} \,,$$

over complexity judgements. The problems $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are called the *sub-problems generated by* Proc *on* $\mathcal{P}$. The processor Proc is *sound* if $\vdash \mathcal{P}\colon f$ is valid whenever the statements $\vdash \mathcal{P}_1\colon f_1, \ldots, \vdash \mathcal{P}_n\colon f_n$ are valid. The processor is *complete* if the inverse direction holds.

3. Let empty denote the axiom $\vdash \langle \varnothing/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle : f$ for all TRSs $\mathcal{W}$ and $\mathcal{Q}$, set of terms $\mathcal{T}$ and $f : \mathbb{N} \to \mathbb{N}$. A *complexity proof* (*proof* for short) of a judgement $\vdash \mathcal{P} : f$ is a deduction using sound processors from the axiom empty and *assumptions* $\vdash \mathcal{P}_1 : f_1, \dots, \vdash \mathcal{P}_n : f_n$, in notation $\mathcal{P}_1 : f_1, \dots, \mathcal{P}_n : f_n \vdash \mathcal{P} : f$.

We say that a complexity proof is *closed* if its set of assumptions is empty, otherwise it is *open*. We follow the usual convention and annotate side conditions as premises to inference rules. As stated in the next lemma, soundness of a processor guarantees our formal system is correct. Completeness ensures that a deduction gives asymptotically tight bounds.

▶ **Lemma 3.4.** *If there exists a closed complexity proof* $\vdash \mathcal{P} : f$*, then the judgement* $\vdash \mathcal{P} : f$ *is valid.*

4 Suiting Reduction Orders to Complexity

Maybe the most obvious tools for complexity analysis in rewriting are *reduction orders*, in particular *interpretations*. Consequently these have been used quite early for complexity analysis. For instance, in [9] *polynomial interpretations* are used in a direct setting in order to estimate the runtime complexity analysis of a TRS. On the other hand in [24] *complexity pairs*, that constitute of a reduction order and a corresponding preorder, are employed to estimate the derivational complexity in a relative setting. Relaxing monotonicity requirements on complexity pairs gives rise to a notion of *reduction pair*, so called *safe reduction pairs* [12], that can be used to estimate the runtime complexity of dependency pair problems, cf. [14, 21]. In the following, we introduce $\mathcal{P}$*-monotone complexity pairs*, that give a unified account of the orders given in [9, 24, 14, 21].

We fix a complexity problem $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$. Consider a proper order $\succ$ on terms, and let $\mathsf{G} : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathbb{N}$. Suppose that $\mathsf{G}(s) > \mathsf{G}(t)$ holds whenever $s \xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} t$ and $s \succ t$ holds, for all terms s reachable from $t \in \mathcal{T}$ with a $\mathcal{P}$-derivation ($s \in \to^*_{\mathcal{P}}(\mathcal{T})$). Then $\succ$ is called G*-collapsible* (on $\mathcal{P}$). If in addition $\mathsf{G}(t)$ is asymptotically bounded by a function $f : \mathbb{N} \to \mathbb{N}$ in the size of t for all start terms $t \in \mathcal{T}$, i.e., $\mathsf{G}(t) \in \mathsf{O}(f(|t|))$ for $t \in \mathcal{T}$, we say that $\succ$ *induces* the complexity f on $\mathcal{P}$. In particular polynomial and matrix interpretations [8, 16] are collapsible, and also *recursive path order* [8] are. All these termination techniques have been suitable tamed so that the induced complexity is a polynomial [9, 18, 3].

Consider an order $\succ$ that induces the complexity f on $\mathcal{P}$. If this order includes the relation $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$, the judgement $\vdash \mathcal{P} : f$ is valid. To check the inclusion, as in [24] we consider a pair of orders $(\succsim, \succ)$ where the preorder $\succsim$ and the order $\succ$ are compatible in the sense that $\succsim \cdot \succ \cdot \succsim \subseteq \succ$ holds. It is obvious that when both orders are monotone and stable under substitutions, the assertions $\mathcal{W} \subseteq \succsim$ and $\mathcal{S} \subseteq \succ$ imply $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} \subseteq \succ$ as desired. Guided by the observation that monotonicity is required only on argument positions that can be rewritten in reductions of starting terms, Hirokawa and the second author [14] propose the use of μ*-monotone orders* for runtime complexity analysis. Initially introduced [25] for termination analysis of *context sensitive rewrite systems* [17], the parameter μ denotes a *replacement map*, i.e., a map that assigns to every n-ary function symbol $f \in \mathcal{F}$ a subset of its argument positions: $\mu(f) \subseteq \{1, \dots, n\}$. In the realm of context sensitive rewriting this map governs under which argument positions a rewrite step is allowed, here μ is used to designate which arguments are *usable* for a set of rules $\mathcal{R}$ in $\mathcal{P}$-derivations, i.e., can be rewritten by a rule $l \to r \in \mathcal{R}$ in $\mathcal{P}$-derivations starting from $t \in \mathcal{T}$.

Denote by $\mathcal{P}\mathrm{os}_\mu(t)$ the μ*-replacing positions* in t, defined as $\mathcal{P}\mathrm{os}_\mu(t) := \{\epsilon\}$ if t is a variable, and $\mathcal{P}\mathrm{os}_\mu(t) := \{\epsilon\} \cup \{i{\cdot}p \mid i \in \mu(f) \text{ and } p \in \mathcal{P}\mathrm{os}_\mu(t_i)\}$ for $t = f(t_1, \dots, t_n)$. For a

binary relation $\to$ on terms we denote by $\mathcal{T}_\mu(\to)$ the set of terms t where subterms at non-μ-replacing positions are in normal form: $t \in \mathcal{T}_\mu(\to)$ if for all positions p in t, if $p \notin \mathcal{P}\mathrm{os}_\mu(t)$ then $t|_p \in \mathrm{NF}(\to)$.

▶ **Definition 4.1.** Let $\mathcal{P}$ be a complexity problem with starting terms $\mathcal{T}$ and let $\mathcal{R}$ denote a set of rewrite rules. A replacement map μ is called a *usable replacement map* for $\mathcal{R}$ in $\mathcal{P}$, if $\to^*_{\mathcal{P}}(\mathcal{T}) \subseteq \mathcal{T}_\mu(\xrightarrow[\mathcal{Q}]{}_{\mathcal{R}})$.

Put otherwise, μ denotes a usable replacement map for $\mathcal{R}$ in $\mathcal{P}$ if for any rewrite step $s \xrightarrow[\mathcal{Q}]{}_{\mathcal{R}} t$ with $s \in \to^*_{\mathcal{P}}(\mathcal{T})$ the rewrite position p is μ-replacing. It is undecidable to determine if μ is a usable replacement map for rules $\mathcal{R}$ in $\mathcal{P}$. Exploiting that for runtime complexity starting terms are basic, in [14] good approximations for full and innermost rewriting are given.

▶ **Example 4.2** (Example 3.2 continued)**.** Consider the $\mathcal{P}_\times$-derivation

$$\underline{\mathbf{2}\times\mathbf{1}} \to_{\mathcal{P}_\times} (\underline{\mathbf{1}\times\mathbf{1}})+\mathbf{1} \to_{\mathcal{P}_\times} ((\underline{\mathbf{0}\times\mathbf{1}})+\mathbf{1})+\mathbf{1} \to_{\mathcal{P}_\times} (\underline{\mathbf{0}+\mathbf{1}})+\mathbf{1} \to_{\mathcal{P}_\times} \mathsf{s}(\underline{\mathbf{0}+\mathbf{0}})+\mathbf{1} \to_{\mathcal{P}_\times} \mathbf{1}+\mathbf{1}\,,$$

where redexes are underlined. Here, and also in consecutive examples, we use the notation $\mathbf{n}$ for the numeral $\mathsf{s}(\cdots(\mathsf{s}(0))\cdots)$ with $n \in \mathbb{N}$ occurrences of the constructor s. Observe that if multiplication occurs in a context, then only under the first argument position of addition. This holds even for all $\mathcal{P}_\times$-derivations of basic terms. The map $\mu_\times$, defined by $\mu_\times(+) = \{1\}$ and $\mu_\times(\times) = \mu_\times(\mathsf{s}) = \varnothing$, thus constitutes a usable replacement map for the multiplication rules $\{3, 4\}$ in $\mathcal{P}_\times$. Since the argument position of s is not usable in $\mu_\times$, the last step witnesses that $\mu_\times$ does not designate a usable replacement map for the addition rules $\{1, 2\}$.

We say that an order $\succ$ is *μ-monotone* if it is monotone on μ positions, in the sense that for all function symbols f, if $i \in \mu(f)$ and $s_i \succ t_i$ then $f(s_1, \ldots, s_i, \ldots, s_n) \succ f(s_1, \ldots, t_i, \ldots, s_n)$ holds. The next intermediate lemma follows by a standard induction on the rewrite position, and is central to the definition of *$\mathcal{P}$-monotone complexity pair* defined below.

▶ **Lemma 4.3.** *Let μ be a usable replacement map for $\mathcal{R}$ in $\mathcal{P}$, and let $\succ$ denote a μ-monotone order that is stable under substitutions. If $\mathcal{R} \subseteq \succ$ holds, i.e., rewrite rules in $\mathcal{R}$ are oriented from left to right, then $s \xrightarrow[\mathcal{Q}]{}_{\mathcal{R}} t$ implies $s \succ t$ for all terms $s \in \to^*_{\mathcal{P}}(\mathcal{T})$.*

▶ **Definition 4.4** (Complexity Pair, $\mathcal{P}$-monotone)**.**

1. A *complexity pair* is a pair $(\succsim, \succ)$, such that $\succsim$ is a stable preorder and $\succ$ a stable order with $\succsim \cdot \succ \cdot \succsim \subseteq \succ$.
2. Suppose $\succsim$ is $\mu_{\mathcal{W}}$-monotone for a usable replacement map of $\mathcal{W}$ in $\mathcal{P}$, and likewise $\succ$ is $\mu_{\mathcal{S}}$-monotone for a usable replacement map of $\mathcal{S}$ in $\mathcal{P}$. Then $(\succsim, \succ)$ is called *$\mathcal{P}$-monotone*.

▶ **Lemma 4.5.** *Consider a $\mathcal{P}$-monotone complexity pair $(\succsim, \succ)$ such that the order $\succ$ is G-collapsible on $\mathcal{P}$. Further, suppose that $(\succsim, \succ)$ is compatible with $\mathcal{P}$ in the sense that $\mathcal{W} \subseteq \succsim$ and $\mathcal{S} \subseteq \succ$ hold. Then $s \xrightarrow[\mathcal{Q}]{}_{\mathcal{S}/\mathcal{W}} t$ implies $s \succ t$ for all terms $s \in \to^*_{\mathcal{P}}(\mathcal{T})$. In particular, $\mathsf{dh}(t, \xrightarrow[\mathcal{Q}]{}_{\mathcal{S}/\mathcal{W}}) \leqslant \mathsf{G}(t)$ for all $t \in \mathcal{T}$.*

Proof. Consider a $\mathcal{Q}$-restricted relative step $s \xrightarrow[\mathcal{Q}]{}^*_{\mathcal{W}} \cdot \xrightarrow[\mathcal{Q}]{}_{\mathcal{S}} \cdot \xrightarrow[\mathcal{Q}]{}^*_{\mathcal{W}} t$ for $s \in \to^*_{\mathcal{P}}(\mathcal{T})$. Using the assumptions on $(\succsim, \succ)$ and the inclusions $\mathcal{W} \subseteq \succsim$ and $\mathcal{S} \subseteq \succ$ to satisfy the assumptions of Lemma 4.3, we obtain $s \succsim^* \cdot \succ \cdot \succsim^* t$. Hence $s \succ t$ follows by transitivity of $\succsim$ and the inclusion $\succsim \cdot \succ \cdot \succsim \subseteq \succ$.

As a consequence, every rewrite sequence $t = t_0 \xrightarrow[\mathcal{Q}]{}_{\mathcal{S}/\mathcal{W}} t_1 \xrightarrow[\mathcal{Q}]{}_{\mathcal{S}/\mathcal{W}} \cdots$ for $t \in \mathcal{T}$ translates to $\mathsf{G}(t) = \mathsf{G}(t_0) > \mathsf{G}(t_1) > \cdots$, thus $\mathsf{dh}(t, \xrightarrow[\mathcal{Q}]{}_{\mathcal{S}/\mathcal{W}})$ is defined and bounded by $\mathsf{G}(t)$. ◀

As immediate consequence of this lemma, we obtain our first processor.

▶ **Theorem 4.6** (Complexity Pair Processor). *Let $(\succsim, \succ)$ be a $\mathcal{P}$-monotone complexity pair such that $\succ$ induces the complexity f on $\mathcal{P}$. The following processor is sound:*

$$\frac{\mathcal{S} \subseteq \succ \quad \mathcal{W} \subseteq \succsim}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle : f} \ \mathsf{CP} \ .$$

When the set of starting terms is unrestricted only the full replacement map is usable for rules of $\mathcal{P}$. In this case, our notion of complexity pairs collapses to the one given by Zankl and Korp [24]. We emphasise that in contrast to [14], our notion of complexity pair is parameterised in separate replacements for $\succsim$ and $\succ$. By this separation we can restate *(safe) reduction pairs* originally proposed in [12], employed in the dependency pair setting below, as instances of complexity pairs (cf. Lemma 5.9).

A variation of the complexity pair processor, that iteratively orients disjoint subsets of $\mathcal{S}$, occurred first in [24]. The following processor constitutes a straight forward generalisation of [24, Theorem 4.4] to our setting.

▶ **Theorem 4.7** (Decompose Processor [24]). *The following processor is sound:*

$$\frac{\vdash \langle \mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle : f \quad \vdash \langle \mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle : g}{\vdash \langle \mathcal{S}_1 \cup \mathcal{S}_2/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle : f + g} \ \mathsf{decompose} \ .$$

Proof. The lemma follows as $\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}_1 \cup \mathcal{S}_2/\mathcal{W}}) \leqslant \mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}}) + \mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}})$. ◀

In combination with for instance complexity pairs, the decompose processor allows as in [24] *shifting* of rules from the strict to the weak component. This is demonstrated in the following proof, that was automatically found by our complexity prover TCT.

▶ **Example 4.8** (Examples 3.2 and 4.2 continued). Consider the linear polynomial interpretation $\mathcal{A}$ over $\mathbb{N}$ such that $0_{\mathcal{A}} = 0$, $\mathsf{s}_{\mathcal{A}}(x) = x + 1$, $x +_{\mathcal{A}} y = x + y$ and $x \times_{\mathcal{A}} y = x \cdot y + x^2$. Let $\mathcal{P}_4 := \langle \{\mathtt{4}\}/\{\mathtt{1},\mathtt{2},\mathtt{3}\}, \mathcal{R}_\times, \mathcal{T}_\mathsf{b} \rangle$ denote the problem that accounts for the rules $\mathtt{4}\colon\ \mathsf{s}(x) \times y \to (x \times y) + y$ in $\mathcal{P}_\times$. The induced order $>_{\mathcal{A}}$ together with its reflexive closure $\geqslant_{\mathcal{A}}$ forms a $\mathcal{P}_4$-monotone complexity pair $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$ that induces quadratic complexity on $\mathcal{P}_4$. The following depicts a complexity proof $\langle \{\mathtt{1},\mathtt{2},\mathtt{3}\}/\{\mathtt{4}\}, \mathcal{R}_\times, \mathcal{T}_\mathsf{b} \rangle : g \vdash \mathcal{P}_\times : n^2 + g$.

$$\dfrac{\dfrac{\{\mathtt{4}\} \subseteq >_{\mathcal{A}} \quad \{\mathtt{1},\mathtt{2},\mathtt{3}\} \subseteq \geqslant_{\mathcal{A}}}{\vdash \langle \{\mathtt{4}\}/\{\mathtt{1},\mathtt{2},\mathtt{3}\}, \mathcal{R}_\times, \mathcal{T}_\mathsf{b} \rangle : n^2} \ \mathsf{CP} \qquad \vdash \langle \{\mathtt{1},\mathtt{2},\mathtt{3}\}/\{\mathtt{4}\}, \mathcal{R}_\times, \mathcal{T}_\mathsf{b} \rangle : g}{\vdash \mathcal{P}_\times : n^2 + g} \ \mathsf{decompose} \ .$$

The above complexity proof can now be completed iteratively, on the simpler problem $\langle \{\mathtt{1},\mathtt{2},\mathtt{3}\}/\{\mathtt{4}\}, \mathcal{R}_\times, \mathcal{T}_\mathsf{b} \rangle$. Since the complexity of $\mathcal{P}_\times$ is cubic, one has to use a technique beyond quadratic polynomial interpretations here. We remark that the decompose processor finds applications beyond its combination with complexity pairs, for instance TCT uses this processor to separation independent components by analysing the *dependency graph* [14].

5 Dependency Pair Processors

The introduction of *dependency pairs* (*DPs* for short) [1], and its formalisation in the *dependency pair framework* [23], drastically increased power and modularity in termination provers. It is well established that the DP method is unsuitable for complexity analysis. The induced complexity is simply too high [22], in the sense that the complexity of $\mathcal{R}$ is not

suitably reflected in its canonical DP problem. Hirokawa and the second author [12] recover this deficiency with the introduction of *weak dependency pairs*. Crucially, weak dependency pairs group different function calls in right-hand sides, using *compound symbols*.

In this section, we first introduce a notion of *dependency pair complexity problem* (*DP problem* for short), a specific instance of a complexity problem. In Theorem 5.8 and Theorem 5.12 we then introduce the *weak dependency pair* and *dependency tuples* processors, that construct from a runtime complexity problem its canonical DP problem. We emphasise that both processors are conceptually *not* new, weak dependency pairs were introduced in [12], and dependency tuples in [21]. Here, we establish a simulation that also accounts for relative rewrite steps, consequently our processors provide a generalisations of [12, 21].

Consider a signature $\mathcal{F}$ that is partitioned into defined symbols $\mathcal{D}$ and constructors $\mathcal{C}$. Let $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ be a term. For $t = f(t_1,\dots,t_n)$ and $f \in \mathcal{D}$, we set $t^\sharp = f^\sharp(t_1,\dots,t_n)$ where $f^\sharp$ is a new n-ary function symbol called *dependency pair symbol*. For t not of this shape, we set $t^\sharp = t$. The least extension of the signature $\mathcal{F}$ containing all such dependency pair symbols is denoted by $\mathcal{F}^\sharp$. For a set $T \subseteq \mathcal{T}(\mathcal{F},\mathcal{V})$, we denote by $T^\sharp$ the set of marked terms $T^\sharp = \{t^\sharp \mid t \in T\}$. Let $\mathcal{C}_{\mathsf{com}} = \{\mathsf{c}_0, \mathsf{c}_1, \dots\}$ be a countable infinite set of fresh *compound symbols*, where we suppose $\mathsf{ar}(\mathsf{c}_n) = n$. Compound symbols are used to group calls in *dependency pairs for complexity* (*dependency pairs* or *DPs* for short). We define $\textsc{com}(t_1,\dots,t_n) := \mathsf{c}_n(t_1,\dots,t_n)$ where $\mathsf{c}_n \in \mathcal{C}_{\mathsf{com}}$ for $n \neq 1$, for $n = 1$ we set $\textsc{com}(t) := t$.

▶ **Definition 5.1** (Dependency Pair, Dependency Pair Complexity Problem).

1. A *dependency pair* (*DP* for short) is a rewrite rule $l^\sharp \to \textsc{com}(r_1^\sharp,\dots,r_n^\sharp)$ where $l, r_1, \dots, r_n \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and l is not a variable.
2. Let $\mathcal{S}$ and $\mathcal{W}$ be two TRSs, and let $\mathcal{S}^\sharp$ and $\mathcal{W}^\sharp$ be two sets of dependency pairs. A complexity problem $\langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ with $\mathcal{T}^\sharp \subseteq \mathcal{T}_{\mathsf{b}}^\sharp$ is called a *dependency pair complexity problem* (or simply *DP problem*).

We keep the convention that $\mathcal{R},\mathcal{S},\mathcal{W},\dots$ are TRSs over $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the marked version $\mathcal{R}^\sharp,\mathcal{S}^\sharp,\mathcal{W}^\sharp,\dots$ always denote sets of dependency pairs.

▶ **Example 5.2** (Example 3.2 continued). Denote by $\mathcal{S}_\times^\sharp$ the dependency pairs

$$\mathsf{5}\colon\ \mathsf{s}(x) \times^\sharp y \to \mathsf{c}_2((x \times y) +^\sharp y, x \times^\sharp y) \qquad \mathsf{6}\colon\ \mathsf{s}(x) +^\sharp y \to x +^\sharp y\ ,$$

and $\mathcal{T}_{\mathsf{b}}^\sharp$ the set of (marked) basic terms with defined symbols $+^\sharp, \times^\sharp$ and constructors $\mathsf{s}, 0$. Then $\mathcal{P}_\times^\sharp := \langle \mathcal{S}_\times^\sharp/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_{\mathsf{b}}^\sharp\rangle$, where $\mathcal{R}_\times$ are the rules for addition and multiplication depicted in Example 3.2, is a DP problem. We anticipate that the DP problem $\mathcal{P}_\times^\sharp$ reflects the complexity of our multiplication problem $\mathcal{P}_\times$, compare Theorem 5.12 below.

For the remainder of this section, we fix a DP problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. We call an n-holed context C a *compound context* if it contains only compound symbols. Consider the $\mathcal{P}_\times^\sharp$ derivation

$$\begin{aligned} D\colon\ \underline{\mathbf{2} \times^\sharp \mathbf{1}} &\to_{\mathcal{P}_\times^\sharp} \mathsf{c}_2((\underline{\mathbf{1} \times \mathbf{1}}) +^\sharp \mathbf{1}, \mathbf{1} \times^\sharp \mathbf{1}) \\ &\to^*_{\mathcal{P}_\times^\sharp} \mathsf{c}_2(\underline{\mathbf{1} +^\sharp \mathbf{1}}, \underline{\mathbf{1} \times^\sharp \mathbf{1}}) \\ &\to^2_{\mathcal{P}_\times^\sharp} \mathsf{c}_2(\mathbf{0} +^\sharp \mathbf{1}, \mathsf{c}_2((\mathbf{0} \times \mathbf{1}) +^\sharp \mathbf{1}, \mathbf{0} \times^\sharp \mathbf{1}))\ . \end{aligned}$$

Observe that any term in the above sequence can be written as $C[t_1,\dots,t_n]$ where C is a maximal compound context, and $t_1,\dots,t_n$ are marked terms without compound symbols. For instance, the last term in this sequence is given as $C[\mathbf{0} \times^\sharp \mathbf{1}, (\mathbf{0} \times \mathbf{1}) +^\sharp \mathbf{1}, \mathbf{0} \times^\sharp \mathbf{1}]$ for

$C := \mathsf{c}_2(\Box, \mathsf{c}_2(\Box, \Box))$. This holds even in general, with the exception that $t_1, \dots, t_n$ are not necessarily marked. Note that such an unmarked term t_i ($i \in \{1, \dots, n\}$) can only result from the application of a collapsing rule $l^\sharp \to x$ for x a variable, which is permitted by our formulation of dependency pair. We capture this observation with the set $\mathcal{T}^\sharp_\to$, defined as the least extension of $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\mathcal{T}^\sharp(\mathcal{F}, \mathcal{V})$ that is closed under compound contexts. Then the following observation holds.

▶ **Lemma 5.3.** *For every TRS $\mathcal{R}$ and DPs $\mathcal{R}^\sharp$, we have $\to^*_{\mathcal{R}^\sharp \cup \mathcal{R}}(\mathcal{T}^\sharp_\to) \subseteq \mathcal{T}^\sharp_\to$. In particular, $\to^*_{\mathcal{P}^\sharp}(\mathcal{T}^\sharp) \subseteq \mathcal{T}^\sharp_\to$ follows.*

Proof. Let $s = C[s_1, \dots, s_n] \in \mathcal{T}^\sharp_\to$ where C is a maximal compound context. Suppose $s \to_{\mathcal{R}^\sharp \cup \mathcal{R}} t$. Since C contains only compound symbols, it follows that $t = C[s_1, \dots, t_i, \dots, s_n]$ where $s_i \to_{\mathcal{R}^\sharp \cup \mathcal{R}} t_i$ for some $i \in \{1, \dots, n\}$, where again $t_i \in \mathcal{T}^\sharp_\to$. Consequently, $t \in \mathcal{T}^\sharp_\to$ and the first half of the lemma follows by inductive reasoning. From this the second half of the lemma follows, using that $\mathcal{T}^\sharp \subseteq \mathcal{T}^\sharp_\to$ and taking $\mathcal{R}^\sharp := \mathcal{S}^\sharp \cup \mathcal{W}^\sharp$ and $\mathcal{R} := \mathcal{S} \cup \mathcal{W}$. ◀

Consider a term $t = C[t_1, \dots, t_n] \in \mathcal{T}^\sharp_\to$ for a maximal compound context C. Any reduction of t consists of *independent sub-derivations* of t_i ($i = 1, \dots, n$), which are possibly interleaved. To avoid reasoning up to permutations of rewrite steps, we introduce a notion of *derivation tree* that disregards the order of parallel steps under compound contexts.

A (directed) *hypergraph* over *labels* $\mathcal{L}$ is a triple $G = (N, E, \mathsf{lab})$ where N is a set of *nodes*, $E \subseteq N \times \mathcal{P}(N)$ a set of *edges*, and $\mathsf{lab} : N \cup E \to \mathcal{L}$ a *labeling function*. For $e = \langle u, \{v_1, \dots, v_n\}\rangle \in E$ we call the node u the *source*, and nodes $v_1, \dots, v_n$ the *targets* of e. We keep the convenience that every node is the source of at most one edge. We denote by $\rightharpoonup_G$ the *successor relation* in G, i.e., $u \rightharpoonup_G v$ if there exists an edge $e = \langle u, \{v_1, \dots, v_n\}\rangle \in E$ with $v \in \{v_1, \dots, v_n\}$. We set $u \xrightarrow{\mathcal{K}}_G v$ if additionally $\mathsf{lab}(e) \in \mathcal{K}$ holds, and abbreviate $\xrightarrow{\{l\}}_G$ by $\xrightarrow{l}_G$. If there exists a *path* $u = w_1 \rightharpoonup_G \cdots \rightharpoonup_G w_n = v$ we say that v is *reachable* from u in G. We call G a *hypertree* (*tree* for short) if there exists a unique node $u \in N$, the *root* of G, such that every $v \in N$ is reachable from u by a unique path.

▶ **Definition 5.4.** Let $t \in \mathcal{T}^\sharp(\mathcal{F}, \mathcal{V}) \cup \mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of $\mathcal{P}^\sharp$ *derivation trees of* t, in notation $\mathsf{DTree}_{\mathcal{P}^\sharp}(t)$, is defined as the least set of labeled hypertrees such that:

1. $T \in \mathsf{DTree}_{\mathcal{P}^\sharp}(t)$ where T consists of a unique node labeled by t.
2. Suppose $t \xrightarrow{\mathcal{Q}}_{\{l \to r\}} \mathrm{COM}(t_1, \dots, t_n)$ for $l \to r \in \mathcal{P}^\sharp$ and let $T_i \in \mathsf{DTree}_{\mathcal{P}^\sharp}(t_i)$ for $i = 1, \dots, n$. Then $T \in \mathsf{DTree}_{\mathcal{P}^\sharp}(t)$, where T is a tree with children T_i ($i = 1, \dots, n$), the root of T is labeled by t, and the edge from the root of T to its children is labeled by $l \to r$.

Figure 1 depicts a derivation tree T of $\mathcal{P}^\sharp_\times$ (cf. Example 5.2) that *corresponds* to the derivation D given below Example 5.2, in the sense that every edge $e = \langle u, \{v_1, \dots, v_n\}\rangle$ in T labeled by rule $l \to r$ corresponds to a rewrite step $t \xrightarrow{\mathcal{Q}}_{l \to r} \mathrm{COM}(t_1, \dots, t_n)$ in D, with t and $t_1, \dots, t_n$ precisely the label of source u and targets $v_1, \dots, v_n$ respectively. We also say that $l \to r$ was *applied* at node u in T. This correspondence leads to the following characterisation of the complexity function of DP problems $\mathcal{P}^\sharp$. Let $|T|_{\mathcal{R}^\sharp \cup \mathcal{R}}$ denote the number of applications of a rule $l \to r$ in the derivation tree T, i.e., the number of edges in T labeled by a rule $l \to r \in \mathcal{R}^\sharp \cup \mathcal{R}$.

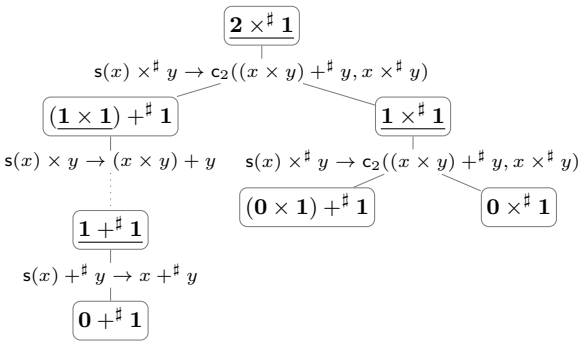


Figure 1 $\mathcal{P}^\sharp_\times$ derivation tree of $\mathbf{2} \times^\sharp \mathbf{1}$.

▶ **Lemma 5.5.** *For every $t \in \mathcal{T}(\mathcal{F},\mathcal{V}) \cup \mathcal{T}^\sharp(\mathcal{F},\mathcal{V})$, we have*

$$\mathsf{dh}(t, \xrightarrow{\mathcal{Q}}_{\mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}}) \simeq \max\{|T|_{\mathcal{S}^\sharp \cup \mathcal{S}} \mid T \text{ is a } \mathcal{P}^\sharp\text{-derivation tree of } t\} \ .$$

In particular $\mathsf{cp}_{\mathcal{P}^\sharp}(n) \simeq \max\{|T|_{\mathcal{S}^\sharp \cup \mathcal{S}} \mid T \text{ is a } \mathcal{P}^\sharp\text{-derivation tree of } t \in \mathcal{T} \text{ with } |t| \leqslant n\}$ *holds.*

5.1 Weak Dependency Pairs and Dependency Tuples

▶ **Definition 5.6** (Weak Dependency Pairs [12])**.** Let $\mathcal{R}$ denote a TRS such that the defined symbols of $\mathcal{R}$, i.e., roots of left-hand sides, are included in $\mathcal{D}$. Consider a rule $l \to C[r_1,\dots,r_n]$ in $\mathcal{R}$, where C is a maximal context containing only constructors. The dependency pair $l^\sharp \to \mathrm{COM}(r_1^\sharp,\dots,r_n^\sharp)$ is called a *weak dependency pair* of $\mathcal{R}$, in notation $\mathsf{WDP}(l \to r)$. We denote by $\mathsf{WDP}(\mathcal{R}) := \{\mathsf{WDP}(l \to r) \mid l \to r \in \mathcal{R}\}$ the set of all weak dependency pairs of $\mathcal{R}$.

In [12] it has been shown that for any term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $\mathsf{dh}(t, \to_{\mathcal{R}}) = \mathsf{dh}(t^\sharp, \to_{\mathsf{WDP}(\mathcal{R})\cup\mathcal{R}})$. We extend this result to our setting, where the following lemma serves as a preparatory step.

▶ **Lemma 5.7.** *Let $\mathcal{R}$ and $\mathcal{Q}$ be two TRSs, such that the defined symbols of $\mathcal{R}$ are included in $\mathcal{D}$. Then every derivation*

$$t = t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_1} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_2} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_3} \cdots \ ,$$

for basic term t and $\mathcal{R}_i \subseteq \mathcal{R}$ $(i \geqslant 1)$ is simulated step-wise by a derivation

$$t^\sharp = s_0 \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}_1)\cup\mathcal{R}_1} s_1 \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}_2)\cup\mathcal{R}_2} s_2 \xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{R}_3)\cup\mathcal{R}_3} \cdots \ ,$$

and vice versa.

▶ **Theorem 5.8** (Weak Dependency Pair Processor)**.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ such that all defined symbols in $\mathcal{S} \cup \mathcal{W}$ occur in $\mathcal{D}$. The following processor is sound and complete.*

$$\frac{\vdash \langle \mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle \colon f}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle \colon f}\ \text{Weak Dependency Pairs}$$

Proof. Set $\mathcal{P} := \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ and $\mathcal{P}^\sharp := \langle \mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$. Suppose first $\mathsf{cp}_{\mathcal{P}^\sharp} \in \mathsf{O}(f(n))$. Lemma 5.7 shows that every $\to_{\mathcal{P}}$ reduction of $t \in \mathcal{T}$ is simulated by a corresponding $\to_{\mathcal{P}^\sharp}$ reduction starting from $t^\sharp \in \mathcal{T}^\sharp$. Observe that every $\xrightarrow{\mathcal{Q}}_{\mathcal{S}}$ step in the considered derivation is simulated by a $\xrightarrow{\mathcal{Q}}_{\mathsf{WDP}(\mathcal{S})\cup\mathcal{S}}$ step. We thus obtain $\mathsf{cp}_{\mathcal{P}} \in \mathsf{O}(f(n))$. This proves soundness, completeness is obtained dual. ◀

Unlike for termination analysis, one has to account also for rewrite rules beside dependency pairs. In contrast, DP problems of the form $\langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ are usually easier to analyse, as rules that need to be accounted for, viz those appearing in the strict component, can only be applied in compound contexts. Some processors tailored for DP problems are even sound only in this setting [6]. Notably, in this setting the complexity pair processor requires that the strict order is monotone only on argument positions of compound symbols:

▶ **Lemma 5.9.** *Let μ denote a usable replacement map for dependency pairs $\mathcal{R}^\sharp$ in $\mathcal{P}^\sharp$. Then μ_{COM} is a usable replacement map for $\mathcal{R}^\sharp$ in $\mathcal{P}^\sharp$, where μ_{COM} denotes the restriction of μ to compound symbols in the following sense: $\mu_{\mathrm{COM}}(\mathsf{c}_n) := \mu(\mathsf{c}_n)$ for all $\mathsf{c}_n \in \mathcal{C}_{\mathsf{com}}$, and otherwise $\mu_{\mathrm{COM}}(f) := \varnothing$ for $f \in \mathcal{F}^\sharp$.*

Proof. For a proof by contradiction, suppose μ_{COM} is not a usable replacement map for $\mathcal{R}^\sharp$ in $\mathcal{P}$. Thus there exists $s \in \to_{\mathcal{P}}^*(\mathcal{T})$ and position $p \in \mathcal{P}\text{os}(s)$ such that $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}^\sharp,p} t$ for some term t, but $p \notin \mathcal{P}\text{os}_{\mu_{\text{COM}}}(s)$. Since $s \in \mathcal{T}^\sharp_\to$ by Lemma 5.3, symbols above position p in s are compound symbols, and so $p \notin \mathcal{P}\text{os}_\mu(s)$ by definition of μ_{COM}. This contradicts however that μ is a usable replacement map for $\mathcal{R}^\sharp$ in $\mathcal{P}$. ◀

We remark that using Lemma 5.9 together with Theorem 4.6, our notion of $\mathcal{P}$-monotone complexity pair generalises *safe reduction pairs* from [12], that constitute of a rewrite preorder $\succsim$ compatible with a total order $\succ$ that is stable under substitutions. Here *safe* means that $\succ$ is monotone on compound contexts. It also generalises the notion of *μ-monotone complexity pair* from [14], that is parameterised by a single replacement map μ for all rules in $\mathcal{P}$.

In [12], the *weight gap principle* is introduced, with the objective to move the strict rules $\mathcal{S}$ into the weak component, in order to obtain a DP problem of the form $\langle \mathcal{S}^\sharp/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$, after the weak dependency pair transformation. *Dependency tuples* introduced in [21] avoid the problem altogether. A complexity problem is directly translated into this form, at the expense of completeness and a more complicated set of dependency pairs.

▶ **Definition 5.10** (Dependency Tuples [21])**.** Let $\mathcal{R}$ denote a TRS such that the defined symbols of $\mathcal{R}$ are included in $\mathcal{D}$. For a rewrite rule $l \to r \in \mathcal{R}$, let $r_1, \dots, r_n$ denote all subterms of the right-hand side whose root symbol is in $\mathcal{D}$. The dependency pair $l^\sharp \to \text{COM}(r_1^\sharp, \dots, r_n^\sharp)$ is called a *dependency tuple* of $\mathcal{R}$, in notation $\mathsf{DT}(l \to r)$. We denote by $\mathsf{DT}(\mathcal{R}) := \{\mathsf{DT}(l \to r) \mid l \to r \in \mathcal{R}\}$, the set of all dependency tuples of $\mathcal{R}$.

The central theorem of [21] states that dependency tuples are sound for runtime complexity analysis. We extend this result to a relative setting.

▶ **Lemma 5.11.** *Let $\mathcal{R}$ and $\mathcal{Q}$ be two TRSs, such that the defined symbols of $\mathcal{R}$ are included in $\mathcal{D}$, and such that* $\text{NF}(\mathcal{Q}) \subseteq \text{NF}(\mathcal{R})$*. Then every derivation*

$$t = t_0 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_1} t_1 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_2} t_2 \xrightarrow{\mathcal{Q}}_{\mathcal{R}_3} \cdots ,$$

for basic term t and $\mathcal{R}_i \subseteq \mathcal{R}$ $(i \geqslant 1)$ is simulated step-wise by a derivation

$$t^\sharp = s_0 \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_1)\cup\mathcal{R}_1} s_1 \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_2)\cup\mathcal{R}_2} s_2 \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R}_3)\cup\mathcal{R}_3} \cdots .$$

▶ **Theorem 5.12** (Dependency Tuple Processor)**.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ be an* innermost *complexity problem such that all defined symbols in $\mathcal{S} \cup \mathcal{W}$ occur in $\mathcal{D}$. The following processor is sound.*

$$\frac{\vdash \langle \mathsf{DT}(\mathcal{S})/\mathsf{DT}(\mathcal{W}) \cup \mathcal{S} \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f}{\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle \colon f}\ \text{Dependency Tuples}$$

Proof. The theorem follows by reasoning identical to Theorem 5.8, using Lemma 5.11. ◀

The problem $\mathcal{P}^\sharp_\times$ depicted in Example 5.2 is obtained from the runtime complexity problem $\mathcal{P}_\times$ of Example 3.2 using the above processor. For the sake of presentation we omitted the trivial dependency pairs $\mathsf{7}\colon 0 +^\sharp y \to \mathsf{c}_0$ and $\mathsf{8}\colon 0 \times^\sharp y \to \mathsf{c}_0$. That this omission is inessential has already been observed in [21], see also the technical report [6] on how this simplification can be formalised in our setting.

6 Dependency Graph Decomposition

In this section we focus on a novel technique that we call *dependency graph decomposition* (*DG decomposition* for short). Our work on this processor is motivated by the fact that we were not aware of a single method that translates a complexity problem into computationally simpler sub-problems, in the sense that any proof is of the form $\mathcal{P}_1 \colon f_1, \dots, \mathcal{P}_n \colon f_n \vdash \mathcal{P} \colon f$ with $f \in \mathsf{O}(f_i)$ for some $i \in \{1, \dots, n\}$. This implies that the maximal bound one can prove is essentially determined by the strength of the employed base techniques, viz complexity pairs. In our experience however, a complexity prover is seldom able to synthesise a suitable and precise complexity pair that induces a complexity bound beyond a cubic polynomial.

We adapt the notion of dependency graph [1] to complexity problems.

▶ **Definition 6.1** (Dependency Graph)**.** Let $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp \cup \mathcal{S}/\mathcal{W}^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ denote a DP problem. The nodes of the *dependency graph* (*DG* for short) $\mathcal{G}$ of $\mathcal{P}^\sharp$ are the dependency pairs from $\mathcal{S}^\sharp \cup \mathcal{W}^\sharp$, and there is an arrow labeled by $i \in \mathbb{N}$ from $s^\sharp \to \textsc{com}(t_1^\sharp, \dots, t_n^\sharp)$ to $u^\sharp \to \textsc{com}(v_1^\sharp, \dots, v_m^\sharp)$ if for some substitutions $\sigma, \tau : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, $t_i^\sharp\sigma \xrightarrow{\mathcal{Q}}{}^*_{\mathcal{S}\cup\mathcal{W}} u^\sharp\tau$.

Figure 2 depicts the dependency graph of our running example $\mathcal{P}_\times^\sharp$, where nodes (5) and (6) refer to the DPs given in Example 5.2. The dependency graph $\mathcal{G}$ indicates in which order dependency pairs can occur in a derivation tree of $\mathcal{P}^\sharp$. To make this intuition precise, we adapt the notion of *DP chain* known from termination analysis to derivation trees. Recall that for a derivation tree T, $\rightharpoonup_T$ denotes the successor relation, and $\xrightarrow{\mathcal{R}}_T$ its restriction to edges labeled by $l \to r \in \mathcal{R}$.

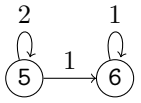


Figure 2 DG of $\mathcal{P}_\times^\sharp$.

▶ **Definition 6.2** (Dependency Pair Chain)**.** Let T be a derivation tree, and consider a path

$$u_1 \xrightarrow{\{l_1 \to r_1\}}_T \cdot \xrightarrow{\mathcal{S}\cup\mathcal{W}}{}^*_T u_2 \xrightarrow{\{l_2 \to r_2\}}_T \cdot \xrightarrow{\mathcal{S}\cup\mathcal{W}}{}^*_T \cdots ,$$

for a sequence of dependency pairs $C \colon l_1 \to r_1, l_2 \to r_2, \dots$. The sequence C is called a *dependency pair chain (in T)*, or *DP chain* for brevity.

▶ **Lemma 6.3.** *Every chain in a $\mathcal{P}^\sharp$ derivation tree is a path in the dependency graph of $\mathcal{P}^\sharp$.*

Dependency graph decomposition seeks to analyse *recursive definitions*, as reflected by *cycles* in the DG, separately. This method is thus closely connected to *cycle analysis* as introduced for termination in [11], that allows the decomposition of the input into separate cycles with respect to the DG.

▶ **Example 6.4** (Example 5.2 continued)**.** Reconsider the problem $\mathcal{P}_\times^\sharp = \langle \{5, 6\}/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp \rangle$ given in Example 5.2. A decomposition into cycles amounts to an inference

$$\frac{\vdash \langle \{5\}/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp \rangle \colon f \quad \vdash \langle \{6\}/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp \rangle \colon g}{\vdash \langle \{5, 6\}/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp \rangle \colon c_{f,g}} ,$$

for cycles (5) and (6), compare Figure 2. This inference is sound for termination analysis [11]. Notice that for f and g we can substitute linear functions, whereas the overall complexity of $\mathcal{P}_\times^\sharp$ is cubic. To see that this bound holds, consider a maximal reduction of $t^\sharp \in \mathcal{T}^\sharp$ for the more involved case $t^\sharp = \mathbf{m} \times^\sharp \mathbf{n}$. Then the i^{th} application of

$$5 \colon \quad \mathsf{s}(x) \times^\sharp y \to \mathsf{c}_2((x \times y) +^\sharp y, x \times^\sharp y) ,$$

in this derivation triggers an independent sub-derivation starting from $t_i^\sharp = \mathbf{m_i} +^\sharp \mathbf{n}$, where $m_i := (m - i) * n$. It is not difficult to verify that the number of applications of (6) in a

sub-derivation of $t_i^\sharp$ is bounded by m_i, thus bounded by a *quadratic* polynomial in the size of $t^\sharp$. As there are at most as many such sub-derivations as there are applications of the DP (5), viz linearly many in the size of $t^\sharp$, we obtain an overall *cubic* bound.

Dependency graph decomposition can infer this bound automatically, using similar reasoning. Taking the call-structure between cycles into account is crucial for such an analysis:

▶ **Example 6.5.** Let $\mathcal{P}^\sharp_{\mathsf{exp}} := \langle \mathcal{R}^\sharp_{\mathsf{exp}}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}_{\mathsf{b}}^\sharp\rangle$ where dependency pairs $\mathcal{R}^\sharp_{\mathsf{exp}}$ are

$$\mathtt{9}\colon \mathsf{d}^\sharp(\mathsf{s}(x)) \to \mathsf{d}^\sharp(x) \qquad \mathtt{10}\colon \mathsf{e}^\sharp(\mathsf{s}(x)) \to \mathsf{c}_2(\mathsf{d}^\sharp(\mathsf{e}(x)), \mathsf{e}^\sharp(x)) \;,$$

and the rewrite system $\mathcal{R}_{\mathsf{exp}}$ is given by the four rules

$$\mathtt{11}\colon \mathsf{d}(0) \to 0 \qquad \mathtt{12}\colon \mathsf{d}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{d}(x))) \qquad \mathtt{13}\colon \mathsf{e}(0) \to 0 \qquad \mathtt{14}\colon \mathsf{e}(\mathsf{s}(x)) \to \mathsf{d}(\mathsf{e}(x)) \;,$$

that compute exponentiation on numerals. The DG of $\mathcal{P}^\sharp_{\mathsf{exp}}$ consists of two cycles, (9) and (10) respectively. While the complexity of $\langle\{\mathtt{9}\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}_{\mathsf{exp}}, \mathcal{T}_{\mathsf{b}}^\sharp\rangle$ and $\langle\{\mathtt{10}\}/\mathcal{R}_{\mathsf{exp}}, \mathcal{R}^\sharp_{\mathsf{exp}}, \mathcal{T}_{\mathsf{b}}^\sharp\rangle$ is again linear, the complexity of $\mathcal{P}^\sharp_{\mathsf{exp}}$ is exponential.

In contrast to a full decomposition into all cycles, DG decomposition produces a pair of sub-problems, obtained by separating the dependency graph between maximal cycles. Iterated application then extends to a separate analysis of all cycles. Call a set of DPs $\mathcal{R}^\sharp$ *forward closed* in $\mathcal{P}^\sharp$, if it is closed under successors with respect to the DG of $\mathcal{P}^\sharp$, i.e., if there is an edge from $s \to t \in \mathcal{R}^\sharp$ to $u \to v$ then also $u \to v \in \mathcal{R}^\sharp$. Throughout the following, we fix a complexity problem $\mathcal{P}^\sharp = \langle \mathcal{S}^\sharp_u \cup \mathcal{S}^\sharp_l \cup \mathcal{S}/\mathcal{W}^\sharp_u \cup \mathcal{W}^\sharp_l \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ whose strict and weak dependency pairs are partitioned such that $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ is *forward closed* in $\mathcal{P}^\sharp$.

As $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ is forward closed in $\mathcal{P}^\sharp$, DPs from $\mathcal{S}^\sharp_u \cup \mathcal{W}^\sharp_u$ can trigger applications of DPs from $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ but not vice versa, compare Lemma 6.3. To formalise this observation, consider a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$. Then the forward closed set $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ induces a separation of T into two (possibly empty) layers, demarcated by topmost applications of DPs from $\mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$: the *lower layer* constitutes of the (maximal) subtrees $T_1, \dots, T_m$ of T with a dependency pair $l \to r \in \mathcal{S}^\sharp_l \cup \mathcal{W}^\sharp_l$ applied at the root, by forward closure these are $\langle \mathcal{S}^\sharp_u \cup \mathcal{S}/\mathcal{W}^\sharp_u \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp\rangle$ derivation trees of some terms $t_i^\sharp$ $(i = 1, \dots, m)$ in T; the *upper layer* consists of the derivation tree $T_\uparrow$ obtained from T by removing the sub-trees $T_1, \dots, T_m$. Compare Figure 3 that illustrates this separation. The DG decomposition processor uses the DPs $\mathsf{sep}(\mathcal{S}^\sharp_u \cup \mathcal{W}^\sharp_u)$, defined as follows, to extend the derivation trees T_i of $t_i^\sharp$ to derivation trees of $t^\sharp \in \mathcal{T}^\sharp$.

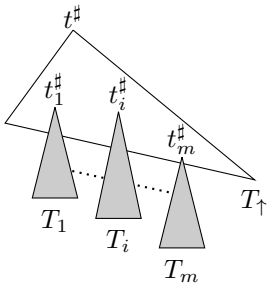


Figure 3 Separation of derivation tree T in upper and lower layer.

▶ **Definition 6.6.** For a set of DPs $\mathcal{R}^\sharp$ we define

$$\mathsf{sep}(\mathcal{R}^\sharp) := \{l \to r_i \mid l \to \mathrm{COM}(r_1, \dots, r_i, \dots, r_k) \in \mathcal{R}^\sharp\} \;.$$

▶ **Example 6.7** (Example 6.4 continued)**.** Consider the complexity problem $\mathcal{P}^\sharp_\times$ from Example 5.2, where $\{\mathtt{6}\colon \mathsf{s}(x) +^\sharp y \to x +^\sharp y\}$ constitutes a forward closed set of DPs with respect to the DG drawn in Figure 2.

Let T denote a $\mathcal{P}^\sharp_\times$ derivation tree of $t^\sharp := \mathbf{m} \times^\sharp \mathbf{n}$ $(m, n \in \mathbb{N})$. For $m_i := (m - i) \cdot n$ $(i = 1, \dots, m)$, the nodes labeled by $t_i^\sharp := \mathbf{m_i} +^\sharp \mathbf{n}$ demarcate upper and lower layer in T, compare the derivation tree depicted in Figure 1. Consider the DPs $\mathsf{sep}(\{\mathtt{5}\})$ given by

$$\mathtt{5a}\colon \mathsf{s}(x) \times^\sharp y \to (x \times y) +^\sharp y \qquad \mathtt{5b}\colon \mathsf{s}(x) \times^\sharp y \to x \times^\sharp y \;.$$

Let T_i $(i = 1, \dots, m)$ denote the sub-trees rooted at the nodes labeled by $t_i^\sharp$ that constitute the lower layer in T. In combination with rewrite rules $\mathcal{R}_\times$, the DPs (5a) and (5b) generate exactly the terms $t_i^\sharp$ from $t^\sharp$. As a consequence, the complexity problem $\langle \{6\}/\{5a, 5b\} \cup \mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_b^\sharp \rangle$ accounts for applications of $\{6\}$ in the sub-trees T_i. In other words, it accounts for applications of DPs in the sub-derivations of $t_i^\sharp$ as investigated in Example 6.4. In correspondence to Example 6.4, it is not difficult to verify that $\vdash \langle \{6\}/\{5a, 5b\} \cup \mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_b^\sharp \rangle \colon n^2$ is valid.

It is also not difficult to verify that $\vdash \langle \{5\}/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_b^\sharp \rangle \colon n$ holds, and this linear bound can be used to bind the applications of the remaining DP (5) in the upper layer $T_\uparrow$ of T, thus in T. As this also bind the number of sub-trees $T_1, \dots, T_m$ that constitute lower layer, we overall get a cubic bound on applications on DPs in T in the size of $t^\sharp$, i.e., $|T|_{\{5,6\}} \in \mathsf{O}(|t^\sharp|^3)$.

The previous complexity proof is an instance of DG decomposition as introduced below. The next two lemmas, used in the soundness proof of the DG decomposition processor, formalise the crucial proof steps employed in Example 6.7. The first observation is simple.

▶ **Lemma 6.8.** *Let $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ be a forward closed set of DPs in $\mathcal{P}^\sharp$, and let T be a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$. Consider the maximal sub-trees $T_1, \dots, T_m$ of T such that $l \to r \in \mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ is applied at the root, and let $T_\uparrow$ be obtained from T by removing $T_1, \dots, T_m$. Then*

1. *$T_\uparrow$ is a $\langle \mathcal{S}_u^\sharp \cup \mathcal{S}/\mathcal{W}_u^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation tree of $t^\sharp$;*
2. *for all $i = 1, \dots, m$, there exists a $\langle \mathcal{S}_l^\sharp \cup \mathcal{S}/\mathcal{W}_l^\sharp \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp), \mathcal{Q}, \mathcal{T}^\sharp \rangle$ derivation trees of $t^\sharp$, that contains T_i as sub-tree.*

Denote by $\mathsf{Pre}_\mathcal{G}(l \to r)$ *direct predecessors* of the dependency pair $l \to r$ in the DG $\mathcal{G}$ of $\mathcal{P}^\sharp$, extended to sets of DPs by $\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp) := \cup_{l \to r \in \mathcal{R}^\sharp} \mathsf{Pre}_\mathcal{G}(l \to r)$.

▶ **Lemma 6.9.** *Let $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ be a forward closed set of DPs in $\mathcal{P}^\sharp$, and let T be a $\mathcal{P}^\sharp$ derivation tree T of $t^\sharp \in \mathcal{T}^\sharp$. Let $T_1, \dots, T_m$ denote the maximal sub-trees of T with $l \to r \in \mathcal{R}^\sharp$ applied at the root. There exists a constant $\Delta \in \mathbb{N}$ depending only on $\mathcal{P}^\sharp$ such that $m \leqslant \max\{1, |T|_{\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp) \setminus \mathcal{R}^\sharp} \cdot \Delta\}$.*

Proof. Let Δ be the maximal arity of a compound symbol from $\mathcal{P}^\sharp$, and observe that every node in T has at most Δ successors. Denote by $\{u_1, \dots, u_m\}$ the roots of T_i $(i = 1, \dots, m)$. The non-trivial case is $m > 1$. In this case, each path from the root of T to the nodes $u_i \in \{u_1, \dots, u_m\}$ contains at least one node with a DP applied. Let $\{v_1, \dots, v_n\}$ collect such nodes closest to $\{u_1, \dots, u_m\}$. In particular, we can thus associate to every node $u_i \in \{u_1, \dots, u_m\}$ a node $v_{i'} \in \{v_1, \dots, v_n\}$ and DP $l \to r \in \mathcal{P}^\sharp$ such that $v_{i'} \xrightarrow{\{l_i \to r_i\}}_T \cdot \xrightarrow{\mathcal{S} \cup \mathcal{W}}_T^* u_i$ holds. As $v_{i'}$ has at most Δ successors and $\xrightarrow{\mathcal{S} \cup \mathcal{W}}_T$ is non-branching, it follows that $m \leqslant \Delta \cdot n$. By Lemma 6.3, for $i = 1, \dots, m$ we see $l_i \to r_i \in \mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp)$. As T_i is maximal, $l_i \to r_i \notin \mathcal{R}^\sharp$. Hence $n \leqslant |T|_{\mathsf{Pre}_\mathcal{G}(\mathcal{R}^\sharp) \setminus \mathcal{R}^\sharp}$ and the lemma follows. ◀

▶ **Theorem 6.10** (Dependency Graph Decomposition)**.** *Consider a dependency pair problem $\mathcal{P}^\sharp = \langle \mathcal{S}_u^\sharp \cup \mathcal{S}_l^\sharp \cup \mathcal{S}/\mathcal{W}_u^\sharp \cup \mathcal{W}_l^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle$ such that (i) $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ is forward closed and (ii) $\mathsf{Pre}_\mathcal{G}(\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp) \cap \mathcal{W}_u^\sharp = \varnothing$ for the DG $\mathcal{G}$ of $\mathcal{P}^\sharp$. The following processor is sound.*

$$\frac{\vdash \langle \mathcal{S}_u^\sharp \cup \mathcal{S}/\mathcal{W}_u^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f \quad \vdash \langle \mathcal{S}_l^\sharp \cup \mathcal{S}/\mathcal{W}_l^\sharp \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp) \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon g}{\vdash \langle \mathcal{S}_u^\sharp \cup \mathcal{S}_l^\sharp \cup \mathcal{S}/\mathcal{W}_u^\sharp \cup \mathcal{W}_l^\sharp \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}^\sharp \rangle \colon f * g} \ \mathsf{DG\ decomp.},$$

for all bounding functions f and g such that $f(n) \neq 0$ and $g(n) \neq 0$ for all $n \in \mathbb{N}$.

Proof. Consider a $\mathcal{P}^\sharp$ derivation tree of $t^\sharp \in \mathcal{T}^\sharp$. We tacitly employ the characterisation of complexity function given in Lemma 5.5, and estimate $|T|_{\mathcal{S}_u^\sharp \cup \mathcal{S}_l^\sharp \cup \mathcal{S}}$ by a function in $\mathsf{O}(f * g)$.

Consider the separation of T as induced by forward closure of $\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp$ into the upper layer $T_\uparrow$, and lower layer consisting of the derivation trees T_i of $t_i^\sharp$ $(i = 1, \dots, m)$, as in Figure 3. By Lemma 6.8(2) the trees T_i $(i = 1, \dots, m)$ can be extended to $\langle \mathcal{S}_l^\sharp \cup \mathcal{S}/\mathcal{W}_l^\sharp \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp), \mathcal{Q}, \mathcal{T}^\sharp\rangle$ derivation tree T_i' of $t^\sharp$. In particular, the complexity of $\langle \mathcal{S}_l^\sharp \cup \mathcal{S}/\mathcal{W}_l^\sharp \cup \mathcal{W} \cup \mathsf{sep}(\mathcal{S}_u^\sharp \cup \mathcal{W}_u^\sharp), \mathcal{Q}, \mathcal{T}^\sharp\rangle$ binds applications of $\mathcal{S}_l^\sharp \cup \mathcal{S}$ in T_i, i.e., $|T_i|_{\mathcal{S}_l^\sharp \cup \mathcal{S}} = |T_i'|_{\mathcal{S}_l^\sharp \cup \mathcal{S}}$. Hence $|T_i|_{\mathcal{S}_l^\sharp \cup \mathcal{S}} \in \mathsf{O}(g(|t^\sharp|))$ by the second precondition of the processor. Similar, Lemma 6.8(1) and the first precondition of the processor gives $|T_\uparrow|_{\mathcal{S}_u^\sharp \cup \mathcal{S}} \in \mathsf{O}(f(|t^\sharp|))$. By assumption (ii) and Lemma 6.9 we see $m \leqslant \max\{1, |T|_{\mathsf{Pre}_\mathcal{G}(\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp) \setminus (\mathcal{S}_l^\sharp \cup \mathcal{W}_l^\sharp)}\} \leqslant \max\{1, |T_\uparrow|_{\mathcal{S}_u^\sharp \cup \mathcal{S}}\}$. Putting these bounds together we get

$$\begin{aligned} |T|_{\mathcal{S}_u^\sharp \cup \mathcal{S}_l^\sharp \cup \mathcal{S}} &= |T_\uparrow|_{\mathcal{S}_u^\sharp \cup \mathcal{S}} + \textstyle\sum_{i=1}^m |T_i|_{\mathcal{S}_l^\sharp \cup \mathcal{S}} \\ &\leqslant |T_\uparrow|_{\mathcal{S}_u^\sharp \cup \mathcal{S}} + \max\{1, |T_\uparrow|_{\mathcal{S}_u^\sharp \cup \mathcal{S}}\} \cdot \max_{i=1}^m |T_i|_{\mathcal{S}_l^\sharp \cup \mathcal{S}} \\ &\in \mathsf{O}(f(|t^\sharp|)) + \mathsf{O}(f(|t^\sharp|)) * \mathsf{O}(g(|t^\sharp|)) = \mathsf{O}(f(|t^\sharp|) * f(|t^\sharp|)) \,. \end{aligned}$$

◀

▶ **Example 6.11** (Example 6.7 continued)**.** Reconsider the DP problem $\mathcal{P}_\times^\sharp = \langle \mathcal{S}_\times^\sharp/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp\rangle$. According to Theorem 6.10, the following depicts a sound inference:

$$\frac{\vdash \langle \{\mathsf{5}\}/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp\rangle : f \quad \vdash \langle \{\mathsf{6}\}/\{\mathsf{5a}, \mathsf{5b}\} \cup \mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp\rangle : g}{\vdash \langle \mathcal{S}_\times^\sharp/\mathcal{R}_\times, \mathcal{R}_\times, \mathcal{T}_\mathsf{b}^\sharp\rangle : f * g} \,.$$

It is not difficult to find polynomial interpretations that verify that the sub-problems have linear and quadratic complexity respectively. Overall we thus obtain the (tight) bound $\mathsf{O}(n^3)$, which in turn binds the complexity of $\mathcal{P}_\times$ by Theorem 5.12.

7 Conclusion

We have presented a combination framework for automated polynomial complexity analysis of term rewrite systems. The framework is general enough to reason about both runtime and derivational complexity, and to formulate a majority of the techniques available for proving polynomial complexity of rewrite systems. On the other hand, it is concrete enough to serve as a basis for a modular complexity analyser, as demonstrated by our automated complexity analyser TCT which closely implements the discussed framework.

Besides the combination framework we have introduced the notion of $\mathcal{P}$-monotone complexity pair that unifies the different orders used for complexity analysis in the cited literature. Last but not least, we have presented the dependency graph decomposition processor. This processor is easy to implement, and greatly improves modularity. This is underpinned by the experimental evidence given online[4] that highlights the strength of our framework, and in particular of the dependency graph decomposition processor.

References

1 T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *TCS*, 236(1–2):133–178, 2000.

2 M. Avanzini. POP* and Semantic Labeling using SAT. In *Proc. of ESSLLI 2008/2009 Student Session*, volume 6211 of *LNCS*, pages 155–166. Springer, 2010.

[4] Available online at `http://cl-informatik.uibk.ac.at/software/tct/experiments/tct2/`.

3 M. Avanzini, N. Eguchi, and G. Moser. New Order-theoretic Characterisation of the Polytime Computable Functions. 2012. Submitted to TCS.

4 M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 33–48, 2010.

5 M. Avanzini and G. Moser. Polynomial Path Orders: A Maximal Model. 2012. Submitted to LMCS. Technical Report available at `http://arxiv.org/abs/1209.3793`.

6 M. Avanzini and G. Moser. A Combination Framework for Complexity, Technical Report. *CoRR, cs/CC/1302.0973*, 2013. Available at `http://www.arxiv.org/abs/1302.0973`.

7 M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. of 24th RTA*. LIPIcs, 2013.

8 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

9 G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with Polynomial Interpretation Termination Proof. *JFP*, 11(1):33–53, 2001.

10 U. Dal Lago and S. Martini. On Constructor Rewrite Systems and the Lambda-Calculus. In *Proc. of 36th ICALP*, volume 5556 of *LNCS*, pages 163–174. Springer, 2009.

11 J. Giesl, T. Arts, and E. Ohlebusch. Modular Termination Proofs for Rewriting Using Dependency Pairs. *JSC*, 34:21–58, 2002.

12 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380, 2008.

13 N. Hirokawa and G. Moser. Complexity, Graphs, and the Dependency Pair Method. In *Proc. of 15th LPAR*, pages 652–666, 2008.

14 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. 2012. Submitted to IC, available at `http://arxiv.org/abs/1102.3129`.

15 D. Hofbauer and C. Lautemann. Termination Proofs and the Length of Derivations. In *Proc. of 3rd RTA*, volume 355 of *LNCS*, pages 167–177. Springer, 1989.

16 D. Hofbauer and J. Waldmann. Termination of String Rewriting with Matrix Interpretations. In *Proc. of 17th RTA*, volume 4098 of *LNCS*, pages 328–342. Springer, 2011.

17 S. Lucas. Fundamentals of Context-Sensitive Rewriting. In *Proc. of 22th SOFSEM*, LNCS, pages 405 – 412. Springer, 1995.

18 A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, and H. Zankl. Joint Spectral Radius Theory for Automated Complexity Analysis of Rewrite Systems. In *Proc. of 4th CAI*, volume 6742 of *LNCS*, pages 1–20. Springer, 2011.

19 G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

20 G. Moser, A. Schnabl, and J. Waldmann. Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In *Proc. of the 28th FSTTCS*, LIPIcs, pages 304–315, 2008.

21 L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. of 23rd CADE*, LNAI, pages 422–438. Springer, 2011.

22 A. Schnabl. *Derivational Complexity Analysis Revisited*. PhD thesis, University of Innsbruck, 2012. Available at `http://cl-informatik.uibk.ac.at/research/`.

23 R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, University of Aachen, 2007. Available as Technical Report AIB-2007-17.

24 H. Zankl and M. Korp. Modular Complexity Analysis via Relative Complexity. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 385–400, 2010.

25 H. Zantema. Termination of Context-Sensitive Rewriting. In *Proc. of 8th RTA*, volume 1232 of *LNCS*, pages 172–186. Springer, 1997.

Tyrolean Complexity Tool: Features and Usage*

Martin Avanzini and Georg Moser

Institute of Computer Science,
University of Innsbruck, Austria
{martin.avanzini,georg.moser}@uibk.ac.at

Abstract

The *Tyrolean Complexity Tool*, TCT for short, is an open source complexity analyser for term rewrite systems. Our tool TCT features a majority of the known techniques for the automated characterisation of polynomial complexity of rewrite systems and can investigate derivational and runtime complexity, for full and innermost rewriting. This system description outlines features and provides a short introduction to the usage of TCT.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases program analysis, term rewriting, complexity analysis, automation

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.71

1 Introduction

In order to measure the complexity of a term rewrite system (TRS for short) it is natural to look at the maximal length of derivation sequences—the *derivation length*—as suggested by Hofbauer and Lautemann in [13]. The resulting notion of complexity is called *derivational complexity*. Hirokawa and the second author introduced in [11] a variation, called *runtime complexity*, that only takes *basic* or *constructor-based* terms as start terms into account. The restriction to basic terms allows one to accurately express the complexity of a program through the runtime complexity of a TRS. An investigation into these notions is of particular interest, as both constitute an *invariant cost model* for rewrite systems [7, 3], in the sense that the actual cost of a reduction on a standard model of computation, viz Turing machines, is bounded by a polynomial in the size of the start term and the length of the reduction. In particular, if the consider TRS defines a function and this TRS admits a polynomial bound on its runtime complexity, then the function is polytime computable.

As first observed in [13], it is by now folklore that termination techniques induce a certain bound on the time complexity of rewrite systems. The seminal paper by Bonfante et. al., [6] gives an early account on taming a termination technique to infer *feasible*, viz *polynomial*, bounds. Since then, a wealth of techniques have been introduced specifically to establish polynomial complexity bounds [2, 11, 17, 18, 20, 21, 19, 15, 1, 12], see [16] for an overview. Motivated not only by these theoretical advances, but also by the *annual international termination competition*[1], which features four dedicated complexity categories since 2008, a vast part of this theoretical body has been implemented in dedicated complexity analysers for rewrite systems. For instance, the termination prover AProVE[2] features powerful support

* This work was partially supported by FWF (Austrian Science Fund) project I-603-N18.
[1] http://www.termination-portal.org/wiki/Termination_Competition/.
[2] http://aprove.informatik.rwth-aachen.de/.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 71–80

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


for analysing the innermost runtime complexity of TRSs. `CaT`[3], a variation of the very fast and powerful termination prover TTT2[4], has excellent support to investigate derivational complexity, and also partial support for runtime complexity analysis. The automated complexity analyser `Matchbox/Poly`[5] verifies polynomially bounded derivational complexity.

Our tool TCT, the *Tyrolean Complexity Tool*, is an automated complexity analyser for TRSs in the line of the aforementioned tools. Its distinct feature is that it is currently the only tool that is competitive, and provides dedicated techniques, for both runtime and derivational complexity analysis. TCT is *open-source*, released under the *GNU Lesser General Public License* (*LGPL*) Version 3, and available from

`http://cl-informatik.uibk.ac.at/software/tct/` .

The theoretical framework underlying TCT, which allows for this generality and modularity, is documented in separate work [5]. Here we want to outline the practical aspects of TCT, version 2.0 to be precise. In Section 2 we provide a brief description of the implementation including accompanying libraries. Section 3, where we discuss features and usage of our tool, constitutes the main part of this work. In Section 4 we indicate future work and conclude.

2 Implementation

Our tool is implemented in the strongly typed, lazy functional programming language `Haskell`[6] and compiles on the *Glasgow Haskell Compiler* on *GNU Linux*. The sources consist of about 13,000 lines of code, and additionally 4,000 lines of documentation. Out of the 73 modules, 43 modules are dedicated to the implementation of the various techniques (roughly 56 % of the code), the remaining modules provide the core of TCT and utilities. Our tool makes also use of following `Haskell` libraries, separately available from the TCT homepage[7], that have been specifically developed for TCT.

- `qlogic` provides facilities for dealing with propositional logic, and consists of approximately 3100 lines of code. Notably it defines an interface to SAT-solvers, including routines to efficiently translate Boolean formulas to conjunctive normal form. Also it features support for theories over natural numbers and integers, implemented by *bit-blasting*.
- `termlib` provides term rewriting functionality, and consists of around 2100 lines of code.
- `parfold` is a small library that provides folding capabilities over lists of concurrently evaluated monad actions, a simple but convenient abstraction to concurrent programming.

3 Features and Usage

The Tyrolean Complexity Tool currently features 23 techniques which are available for runtime and, where applicable, for derivational complexity analysis. Our implementation follows closely the framework provided in [5] which ensures that the techniques are implemented in a modular way. We indicate some characteristic methods implemented in TCT:

Matrix Interpretations: Our tool features an implementation of *matrix interpretations* over the naturals [8], as well as *arctic interpretations* [14]. To weaken monotonicity requirements

[3] Available from `http://cl-informatik.uibk.ac.at/software/cat/`.
[4] Available from `http://cl-informatik.uibk.ac.at/software/ttt2/`.
[5] Available from `http://dfa.imn.htwk-leipzig.de/matchbox/poly/`.
[6] An open-source product of more than twenty years of cutting-edge research, c.f. `http://haskell.org/`.
[7] `http://cl-informatik.uibk.ac.at/projects/`.

we have integrated the *usable arguments criterion* [12] and *usable rules w.r.t. argument filterings* [10]. In order to give polynomial bounds on the induced complexity, TCT can employ *triangular matrices* [18], or use the criterions defined in [15, 20]. Moreover, our implementation also integrates the *weight gap principle* [12].

Polynomial Path Orders: Up to our knowledge, TCT is the only tool that features an implementation of *polynomial path orders* [2, 4] as well as the recently introduced *small polynomial path orders* [1]. Both orders constitute a miniaturisation of recursive path orders that induce polynomially bounded innermost runtime complexity. Whereas the former order can only deduce if the innermost runtime complexity is in principle polynomial, its small brother allows a precise control on the complexity certificate obtained.

Match-Bounds: *Match-Bounds* for term rewrite systems [9] is a powerful termination method that induces linear complexity. Our tool supports *match-*, *top-* and *roof-bounds* both for derivational, and its refinement to runtime complexity analysis.

Weak Dependency Pairs and Dependency Tuples: The introduction of *weak dependency pairs* greatly simplifies the task of estimating the runtime complexity of TRSs. Our tool supports this method as well as its refinement to *innermost* rewriting, called *dependency tuples* in [19]. This technique gives rise to advanced techniques specifically designed for the dependency pair setting, notably various *simplifications*, *usable rules*, *path analysis*, *(safe) reduction pairs* as well as *dependency graph decomposition*, compare [12, 19, 5].

In the following, we discuss usage of TCT.

3.1 Web Interface

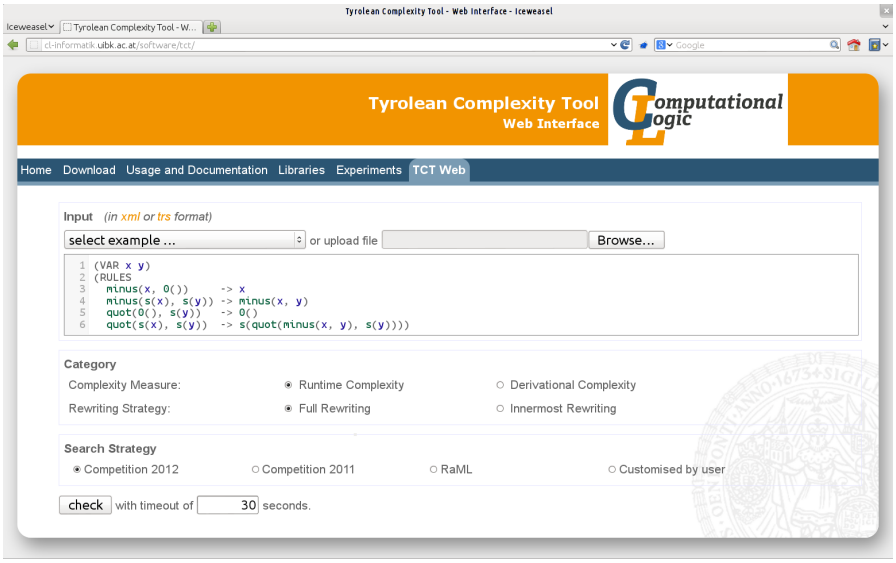


Figure 1 Web Interface of TCT.

Our *web interface*, accessible from the TCT homepage, provides a convenient way to use TCT without the necessity to install the software. The interface is aimed for simplicity, compare Figure 1. For the curious user that wants to play around with TCT, we also provide a wealth of interesting examples. The web interface is configured so that by default an upper bound on the *runtime complexity* of the given rewrite system is estimated. This behaviour can be modified under *category*, where the user can pick from the four different complexity measures TCT currently supports. On success this certificate is presented to the user, together with a proof script that explains in considerable detail how the certificate was obtained.

To find a proof in a reasonable amount of time, the different techniques implemented in TCT need to be combined wisely. This combination depends on the one hand on the input problem, but on the other hand also on the available hardware. In TCT, *proof search* is not hard-wired, instead it is guided by a *(proof) search strategy*. The interface allows to specify such a search strategy from a set of pre-defined proof search strategies. Besides the search strategies employed in recent competitions, the web-interface currently offers the search strategy *RaML*, specifically designed for functional programs given as rewrite systems, and a customisable search strategy that allows the explicit inclusion/exclusion of methods.

3.2 Command-Line Interface

The full power of TCT is available through its command-line interface. For installation instructions we refer the reader to the homepage. Here we want to briefly outline usage and customisation, comprehensive documentation can be found online. TCT is run by

```
$ tct [options] [-s <strategy>] <file>,
```

from the command-line, where `[options]` specify an optional list of command-line options, `<strategy>` specifies optionally a proof search strategy, and `<file>` the *input file*. The input file must adhere either the old *TPDB format*[8] or the new *XTC Format*[9].

A list of options can be obtained by typing `tct --help`. In the command-line interface, the proof search strategy is given as an *S-expression* of the form

```
(<name> [:<argname> <arg>]* [<arg>]*),
```

where outermost parentheses can be dropped. Here `<name>` refers to the name of a proof technique, also called *processor*, the list `[:<argname> <arg>]*` can be used to specify *named optional arguments*, and the list `[<arg>]*` gives a possibly empty sequence of *positional arguments*. As an example,

```
fastest (timeout 3 (bounds :enrichment match)) (matrix :degree 2),
```

provides a valid proof search strategy in TCT. Here `fastest` is used to combine one or more processors in parallel, in this case the `bound` and `matrix` processors, solving the input problem with whichever processor succeeds first. The defined search strategy advises TCT to check for three seconds for *match-boundedness* of the input problem, respectively compatibility with a matrix interpretation that induces a quadratic upper bound. All implemented techniques, including a wealth of *processor combinators* like `fastest` and `timeout`, can be applied directly from the command-line with the option `-s <strategy>`. A complete list of available search strategies, including synopsis and documentation, can be obtained by typing `tct --list`.

Besides basic options given on the command-line, TCT can be configured by modifying the *configuration file*, which resides in `~/.tct/tct.hs` by default. This `Haskell` source-file defines the actual binary that is run each time TCT is called. Thus the full expressiveness of `Haskell` is available; as a downside, it requires also a working `Haskell` environment. A minimal configuration is generated automatically on the first run of TCT. This initial configuration consists of a set of convenient imports and the IO action `main` together with a *configuration record* `config`. The configuration record passed in `main` allows one to overwrite various flags of TCT. Most importantly, through the field `strategies` it also allows the modification of the list of proof search strategies that can be employed.

---

[8] `http://www.lri.fr/~marche/tpdb/format.html`.
[9] `http://www.termination-portal.org/wiki/XTC_Format_Specification`.

```
import Tct (tct)
import Tct.Instances
✂ ..........................................................................................................

main :: IO ()
main = tct config

config :: Config
config = defaultConfig { strategies = strategies } where
  strategies = [ matrices ::: strategy "matrices" ( optional naturalArg "start" (Nat 1) :+: naturalArg )
               , withDP   ::: strategy "withDP" ]

matrices (Nat s :+: Nat n) =
  fastest [ matrix `withDimension` d `withBits` bitsForDimension d | d <- [s..s+n] ] where
    bitsForDimension d = if d < 3 then 2 else 1

withDP =
  (timeout 5 dps <> dts)
  >>> try (exhaustively partitionIndependent)
  >>> try cleanTail
  >>> try usableRules where
    dps = dependencyPairs >>> try usableRules >>> wgOnUsable
    dts = dependencyTuples
    wgOnUsable = weightgap `withDimension` 1 `wgOn` WgOnTrs
```

Figure 2 Configuration defining two new search strategies, called `matrices` and `withDP`.

In Figure 2 we depict a modified configuration that defines two new search strategies, called `matrices` and `withDP`. Strategies are added by overwriting the field `strategies` with a list of declarations of the form

```
<code> ::: strategy "<name>" [<parameters-declaration>] .
```

Here `<code>` refers to a definition that evaluates to a processor, and `"<name>"` as well as the optional parameters-declaration specify how this code is accessible from the command-line. For instance, the first declaration in Figure 2 defines a new search strategy named *matrices*, which is available by supplying the option `-s "matrices [:start <nat>] <nat>"` to the TcT executable. Here the parameters to `matrices` are declared by

```
optional naturalArg "start" (Nat 1) :+: naturalArg ,
```

where the infix operator `:+:` is used to specify sequences of parameters. As indicated by the constructor `naturalArg`, the search strategy `matrices` expect two *natural numbers* as arguments. In contrast to the second parameter, the first is optional and defaults to the natural number `1`.

In Figure 2, these parameters are provided to the code of `matrices`. Using parameters s and n as supplied on the command-line, the code evaluates to a processor that searches for n compatible matrix interpretations of increasing dimension, in parallel. Both `matrix` and `fastest`, along with other processors, combinators and *modifiers* like `withDimension` and `withBits`, are exported by the module `Tct.Instances`.

The second proof search strategy declared in Figure 2 defines a *transformation* called `withDP`. Transformations are a specific class of processors, that generate from the given input problem a possibly empty set of *sub-problems*, in a complexity-preserving manner.[10] For every transformation t and processor p, one can use the *processor* t `>>|` p which first applies transformation t and then solves the resulting sub-problems using p. Search strategy declarations perform such a lifting of transformation implicitly, the declaration of `withDP` for instance results in a search strategy available as `withDP <processor>`. Besides the

[10] Transformations were introduced in Version 1.7 of TcT. Although any processor like `matrix` could also be defined as a transformation, the distinction in TcT is present mainly for historical reasons.

combinator `≫|` and its variation `≫‖`, where the given processor p is applied in parallel on all sub-problems, the module `Tct.Instances` provides a wealth of *transformation combinators*. We briefly discuss here the most important ones. The transformation t_1 `<>` t_2 employed in `withDP` first applies transformation t_1, only if this is unsuccessful it applies transformation t_2 on the input problem instead. A variation of the combinator is given by t_1 `<‖>` t_2 that applies transformations t_1 and t_2 in parallel, resulting in the sub-problems of whichever transformation succeeds first. The combinator `<‖>` thus implements a form of non-deterministic choice. The combinator `⋙` defines composition of transformations, in the sense that the transformation t_1 `⋙` t_2 first applies transformation t_1 and then transformation t_2 on all resulting sub-problems. We remark that any transformation aborts if it is inapplicable. The combinator `try` overrides this behaviour, in the sense that `try` t behaves exactly like t should t succeed, otherwise it behaves as an identity. Finally, the combinator `exhaustively`, defined by `exhaustively` t `= ` t `⋙ try (exhaustively` t`)`, applies t in an iterated fashion.

In total, the defined search strategy `withDP` depicted in Figure 2 applies weak dependency pairs (as realised in the definition of `dps`), or dependency tuples (as realised by `dts`) should the former fail. This transformation is followed by a sequence of syntactic simplifications, if applicable. We remark the thoughtful use of `try`. The transformation `dps` fails if the *weight gap principle* cannot be established on all TRS rules, i.e., rules that are not dependency pairs. The latter is implemented by the transformation `wgOnUsable`, and constitutes an implementation of [12, Theorem 6.5]. We finally point out that an extended version of the transformation `withDP` is available in TCT as `toDP`.

3.3 Interactive Interface

TCT features also an *interactive interface*, TCT-i for short. In this section we guide the reader through a small interactive session that outlines the main features, elaborate documentation of this mode is again provided online.

This *semi-automatic* mode is in particular useful when investigating into tight(er) complexity bounds, and to crack hard-to-solve problems. The interactive interface constitutes essentially of a tiny wrapper around `ghci`, the interpreter bundled with the *Glasgow Haskell compiler*. Users familiar with `ghci` will note that all features available in `ghci` are also available in TCT-i. The interactive interface is started from the command-line by supplying the option `-i` to the TCT executable.

```
$ tct -i
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
✂.....................................................................................................
    This is version 2.0 of the Tyrolean Complexity Tool.

    (c) Martin Avanzini <martin.avanzini@uibk.ac.at>,
        Georg Moser <georg.moser@uibk.ac.at>, and
        Andreas Schnabl <andreas.schnabl@uibk.ac.at>.

    This software is licensed under the GNU Lesser General Public
    License, see <http://www.gnu.org/licenses/>.

    Don't know how to start? Type 'help'.
TCT>
```

The interactive interface maintains a *proof state*, which consists conceptually of a list of *open problems* together with proof information. The command `load` *"<file>"* is used to populate the proof state by the TRS given as argument.

```
TCT> load "examples/div.trs"

    Current Proof State ------------------------------------------------
```

```
Selected Open Problems:
-----------------------
  Strict Trs:
    { -(x, 0()) -> x
    , -(s(x), s(y)) -> -(x, y)
    , %(0(), s(y)) -> 0()
    , %(s(x), s(y)) -> s(%(-(x, y), s(y))) }
  StartTerms: basic terms
  Strategy: none

--------------------------------------------------------------------------------
```

The current state can be inspected at any time by typing the command `state`. We note that the rewrite strategy and set of start terms are defined in accordance to the input file. The commands `set[DC|RC|IDC|IRC]` provide short-hands to these accordingly.

The primary means to modify the proof state is the use of the command `apply`. This command takes a single argument, a transformation or processor respectively, which is applied by default on all open problems collected in the current proof state. Both processors and transformations as imported from `Tct.Instances` qualify as arguments to `apply`. Of course one can also use the various combinators that we have seen so far to construct more complex arguments. Notably, since TCT-i loads the configuration file of TCT, all declarations given in the configuration are available as top-level bindings, and can thus be used in conjuction with `apply`. Recall that our configuration defines a transformation `withDP` that computes weak dependency pairs or dependency tuples respectively, applying various transformations on success. We use this transformation to simplify the input problem.

```
TCT> apply withDP

  Problems simplified. Use 'state' to see the current proof state.
```

The output of `apply` is intentionally kept short.[11] By typing `state` one can observe that our initially loaded complexity problem has been replaced by the problem obtained by our transformation `withDP`. To see that proof generated so far, one can use the command `proof`. Note that as long as the list of open problems is not empty, this proof is marked as open.

```
TCT> proof

  1) dp [OPEN]:
  -------------
    We consider the following problem:
      Strict Trs:
        { -(x, 0()) -> x
        , -(s(x), s(y)) -> -(x, y)
        , %(0(), s(y)) -> 0()
        , %(s(x), s(y)) -> s(%(-(x, y), s(y))) }
      StartTerms: basic terms
      Strategy: none

    We add following weak dependency pairs:
✂..................................................................................................................
    1.1) Open Problem [OPEN]:
    -------------------------

      We consider the following problem:
        Strict DPs:
          { -^#(x, 0()) -> c_1(x)
          , -^#(s(x), s(y)) -> c_2(-^#(x, y))
          , %^#(s(x), s(y)) -> c_4(%^#(-(x, y), s(y))) }
        Weak Trs:
          { -(x, 0()) -> x
          , -(s(x), s(y)) -> -(x, y) }
        StartTerms: basic terms
        Strategy: none
```

[11] To override this behaviour and see actions performed, one can use the command `setShowProofs`, or alternatively set the field `interactiveShowProofs` to `True` in the configuration record of TCT.

Besides `state` and `proof`, following commands allow for further inspection of the current proof state. The command `problems` returns the list of open problems, `wdgs` and `cwdgs` return the corresponding dependency graphs respectively congruence graph and `uargs` returns the usable argument positions. Cf. [12] for an explanation of these attributes. For instance, we can use `cwdgs` to inspect the congruence graph of our open problem as follows.

```
TCT> [cwdg] <- cwdgs
   Congruence Graph of Problem 1:
     ->1:{1,2}
        |
        `->2:{3}

     Here dependency-pairs are as follows:

     Strict DPs:
       { 1: -^#(x, 0()) -> c_1(x)
       , 2: -^#(s(x), s(y)) -> c_2(-^#(x, y))
       , 3: %^#(s(x), s(y)) -> c_4(%^#(-(x, y), s(y))) }
TCT> :module +Tct.Method.DP.DependencyGraph
TCT> isEdgeTo cwdg 1 2
   True
```

Once the list of open problem is empty, the complexity of the input problem has been successfully proven. We can do so on our running example using the matrix processor that we have already used before.

```
TCT> apply matrix
   Hurray, the problem was solved with certicficate YES(O(1),O(n^2)).
   Use 'proof' to show the complete proof.
```

We have found a closed proof that verifies that our initial problem has at most quadratic runtime complexity. We remark that the runtime complexity of the input TRS is even linear. Inspecting the proof we see that the imprecision in the certificate was introduced in the last proof step. Fortunately TCT-i provides a command `undo` that can be used to revert the effect of `apply`. In fact, it reverts any modification on the proof state, except of course the effect of `undo` itself. We refine the proof by restricting the *induced degree* of the constructed interpretation.

```
TCT> undo
   Current Proof State ------------------------------------------------

   Selected Open Problems:
   -----------------------
     Strict DPs:
       { -^#(x, 0()) -> c_1(x)
       , -^#(s(x), s(y)) -> c_2(-^#(x, y))
       , %^#(s(x), s(y)) -> c_4(%^#(-(x, y), s(y))) }
     Weak Trs:
       { -(x, 0()) -> x
       , -(s(x), s(y)) -> -(x, y) }
     StartTerms: basic terms
     Strategy: none

   ------------------------------------------------------------------------

TCT> apply $ matrix `withDegree` Just 1
   Hurray, the problem was solved with certicficate YES(O(1),O(n^1)).
   Use 'proof' to show the complete proof.
```

Here the function `withDegree` is used to modify the default parameters as defined in `matrix`.[12] We finally end up with a closed proof that verifies that our loaded TRS has linear runtime complexity. Using the command `writeProof` *"<file>"* one can write the constructed proof to the given file.

[12] The application operator `$` has low, right-associative binding precedence.

This completes the short tutorial. We remark that for the GNU Emacs[13] enthusiast, we have also crafted a small major-mode for TcT-i. This mode is available in the source distribution of TcT. The mode can be started by typing `M-x tct` into GNU Emacs. In addition to the features explained above, the major-mode provides a refurbished view on the proof state, compare Figure 3 which shows an example session. The approximated dependency graph depicted in Figure 3 is visualised using the `dot` tool of the *Graphviz* toolkit.[14]

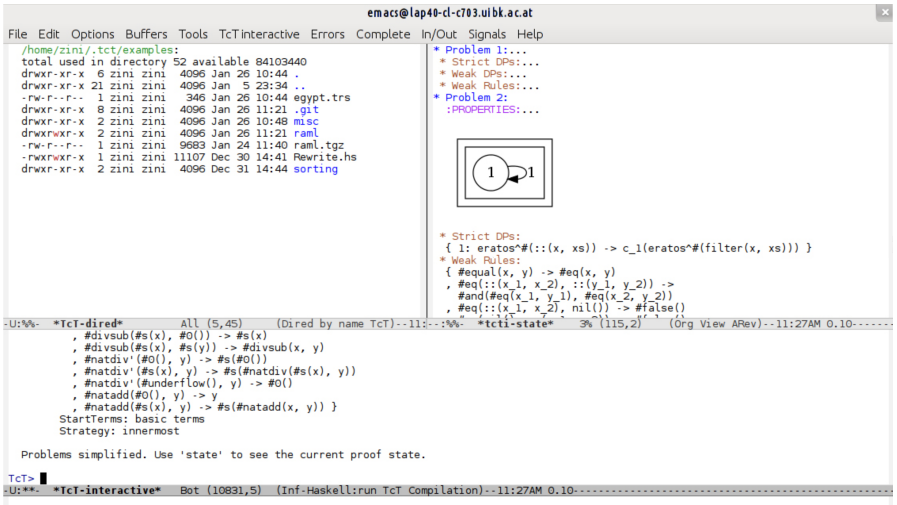


Figure 3 TcT Major Mode for GNU Emacs.

4 Conclusion and Future Work

Our complexity analyser TcT has matured to a state where we can say that it is both versatile and powerful. This is underpinned by the experimental evidence given online[15] which highlights in particular the strength of the underlying combination framework presented in [5].

We of course seek to keep the implementation in line with the active research community. In the upcoming version, we also intend to remove some out-dated design choices, foremost the separation of processors and transformations, which will result in a significantly simplified core. Also, we currently investigate the integration of constrained rewriting. This should leverage the design of complexity preserving reductions from *real world* programs to rewrite systems, in the hope that TcT will act as a powerful backend.

Acknowledgement

Foremostly we thank Andreas Schnabl for his major role in the development of TcT. Moreover, we thank Martin Korp, Christian Sternagel, and Harald Zankl as (former) members of the T$_T$T$_2$ team for ongoing discussions. We also thank Bertram Felgenhauer for valuable discussions concerning the implementation. Finally, we also thank the anonymous reviewers for valuable suggestions that improved the presentation of this paper.

[13] GNU Emacs is open-source and available from `http://www.gnu.org/s/emacs/`.
[14] The toolkit is open-source and available from `http://www.graphviz.org/`.
[15] C.f. `http://cl-informatik.uibk.ac.at/software/tct/experiments/tct2/`.

References

1 M. Avanzini, N. Eguchi, and G. Moser. New Order-theoretic Characterisation of the Polytime Computable Functions. In *Proceedings of the 10th Asian Symposium Programming Languages and Systems*, number 7705 in LNCS, pages 280–295. Springer, 2012.

2 M. Avanzini and G. Moser. Complexity Analysis by Rewriting. In *Proc. of 8th FLOPS*, volume 4989 of *LNCS*, pages 130–146. Springer, 2008.

3 M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 33–48, 2010.

4 M. Avanzini and G. Moser. Polynomial Path Orders: A Maximal Model. 2012. Submitted to LMCS. Technical Report available at `http://arxiv.org/abs/1209.3793`.

5 M. Avanzini and G. Moser. A Combination Framework for Complexity. In *Proc. of 24th RTA*. LIPIcs, 2013. Technical Report available at `http://arxiv.org/abs/1302.0973`.

6 G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with Polynomial Interpretation Termination Proof. *JFP*, 11(1):33–53, 2001.

7 U. Dal Lago and S. Martini. Derivational Complexity is an Invariant Cost Model. In *Proc. of 1st FOPARA*, 2009.

8 J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *JAR*, 40(2-3):195–220, 2008.

9 A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On Tree Automata that Certify Termination of Left-Linear Term Rewriting Systems. In *Proc. of 16th RTA*, number 3467 in LNCS, pages 353–367. Springer, 2005.

10 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *JAR*, 37(3):155–203, 2006.

11 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380, 2008.

12 N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. 2012. Submitted to IC, available at `http://arxiv.org/abs/1102.3129`.

13 D. Hofbauer and C. Lautemann. Termination Proofs and the Length of Derivations. In *Proc. of 3rd RTA*, volume 355 of *LNCS*, pages 167–177. Springer, 1989.

14 A. Koprowski and J. Waldmann. Max/Plus Tree Automata for Termination of Term Rewriting. *AC*, 19(2):357–392, 2009.

15 A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, and H. Zankl. Joint Spectral Radius Theory for Automated Complexity Analysis of Rewrite Systems. In *Proc. of 4th CAI*, volume 6742 of *LNCS*, pages 1–20. Springer, 2011.

16 G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

17 G. Moser and A. Schnabl. Proving Quadratic Derivational Complexities using Context Dependent Interpretations. In *Proc. of 19th RTA*, volume 5117 of *LNCS*, pages 276–290. Springer, 2008.

18 G. Moser, A. Schnabl, and J. Waldmann. Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In *Proc. of 28th FSTTCS*, LIPIcs, pages 304–315, 2008.

19 L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. of 23rd CADE*, LNAI, pages 422–438. Springer, 2011.

20 J. Waldmann. Polynomially Bounded Matrix Interpretations. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 357–372, 2010.

21 H. Zankl and M. Korp. Modular Complexity Analysis via Relative Complexity. In *Proc. of 21st RTA*, volume 6 of *LIPIcs*, pages 385–400, 2010.

Abstract Logical Model Checking of Infinite-State Systems Using Narrowing

Kyungmin Bae[1], Santiago Escobar[2], and José Meseguer[1]

1 **University of Illinois at Urbana-Champaign, IL, USA**
{kbae4,meseguer}@cs.uiuc.edu
2 **Universitat Politècnica de València, Spain**
sescobar@dsic.upv.es

Abstract

A concurrent system can be naturally specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ where states are elements of the initial algebra $\mathcal{T}_{\Sigma/E}$ and concurrent transitions are axiomatized by the rewrite rules R. Under simple conditions, narrowing with rules R modulo equations E can be used to symbolically represent the system's state space by means of terms with logical variables. We call this symbolic representation a *logical state space* and it can also be used for model checking verification of LTL properties. Since in general such a logical state space can be infinite, we propose several abstraction techniques for obtaining either an over-approximation or an under-approximation of the logical state space: (i) a *folding* abstraction that collapses patterns into more general ones, (ii) an easy-to-check method to define (bisimilar) *equational abstractions*, and (iii) an *iterated bounded model checking* method that can detect if a logical state space within a given bound is *complete*. We also show that folding abstractions can be *faithful* for safety LTL properties, so that they do *not* generate any spurious counterexamples. These abstraction methods can be used in combination and, as we illustrate with examples, can be effective in making the logical state space finite. We have implemented these techniques in the Maude system, providing the first narrowing-based LTL model checker we are aware of.

1998 ACM Subject Classification D.2.4 Model Checking

Keywords and phrases model checking, infinite states, rewrite theories, narrowing

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.81

1 Introduction

Model checking of finite-state systems is very well-developed (see. e.g., [2, 8]) and is well-supported by many tools and algorithms. Although model checking of infinite-state systems is more challenging, important advances have been made by various approaches, e.g. [1, 4, 11, 19, 22]. What many of these approaches have in common is the use of *symbolic* representations —such as formulas in a decidable logic or regular (string or tree) languages— to represent not just states but possibly infinite *sets of states*.

An intriguing possibility —first proposed in [31] for the simpler case of reachability analysis, and extended in [16] to LTL model checking— is to use rewriting-based symbolic techniques for infinite-state model checking by: (i) formalizing a concurrent system as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, whose states are elements of the initial algebra $\mathcal{T}_{\Sigma/E}$, and whose concurrent transitions are axiomatized by the rules R; (ii) representing possibly infinite sets of states by Σ-terms $t(x_1, \ldots, x_n)$ with *logical variables* $x_1, \ldots, x_n$, so that $t(x_1, \ldots, x_n)$ describes the set of its ground instances modulo E; and (iii) assuming an E-unification algorithm is available, exploring the *logical state space*, whose states are terms with logical variables $t(x_1, \ldots, x_n)$, by performing *narrowing with R modulo E* (see Section 2).

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 81–96

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


However, at the time the papers [16, 31] were published, no implementation of such narrowing-based logical model checking existed, and important open problems had to be resolved before its practical effectiveness could be demonstrated. This paper is all about solving such open problems by developing new narrowing-based model checking methods, and incorporating these new methods in an actual tool that can demonstrate the practical effectiveness of the narrowing-based model checking approach.

Open Problems Addressed. The main problems left unresolved by [16, 31] include:

1. *Dealing with infinite logical state spaces.* Narrowing can in general generate an infinite number of symbolic states; even though the idea of *folding* logical states by means of the subsumption $\preccurlyeq_E$ modulo E proposed in [16] showed that an infinite logical state space could sometimes be folded into a finite one, this was just one method, and no model checking algorithm existed when it failed.
2. *Dealing with a broad class of theories for which finitary unification algorithms exist.* The key point is that the equations E in a rewrite theory $\mathcal{R}$ must define not just structural axioms of the state, such as the associative-commutative nature of a set of processes, for which well-known unification algorithms exist: E must also define the truth values of the *state predicates* on which the temporal logic formulas are based. There is no hope that a finite set of unification algorithms can handle equations E of this kind: generic methods that can support a broad class of user-defined equational theories E are needed.
3. *Dealing with spurious counterexamples.* The use of abstractions typically brings with it spurious counterexamples that violate the given LTL formula on the abstract system but not in the concrete one. Can this spuriousness be avoided?

Our Contributions. Problem (1) is addressed in a twofold way by: (i) developing new abstraction techniques for logical state spaces that can be seamlessly combined with folding, such as *equational abstractions* (extended and simplified from their use in concrete state spaces in [30]) and the new *bisimilar equational abstractions* (Section 3.3); and (ii) developing a new *bounded* LTL *logical model checking algorithm* that does not require the state space to be finite, can model check a system up to a given depth, and can detect that a finite state space exists within the depth and support full verification in that case (Section 4).

Problem (2) is addressed by supporting equational theories E having the *finite variant property* [10] using the generic variant narrowing and unification algorithms in [17]; our approach is similar to that of the Maude-NPA [15], but is applied here to general LTL model checking, whereas Maude-NPA only supports reachability analysis for a restricted domain.

Problem (3) is addressed by showing that folding the logical state space by means of the subsumption $\preccurlyeq_E$ modulo E can give a *faithful* abstraction that does *not* generate any spurious counterexamples when verifying *safety LTL* properties (Section 3.2). This faithfulness of course holds when folding with $\preccurlyeq_E$ and bisimilar equational abstractions are used in combination. Note that folding abstractions are *strictly* more general than bisimulations since they are *not* faithful for general LTL properties.

Another important contribution is that all these new methods are supported by the new Maude LTL logical model checker that uses the Maude infrastructure [9, 12] for variant narrowing/unification and has many of its features implemented at the C++ level for efficiency reasons. We illustrate both the effectiveness of the tool and the new methods presented here by means of two nontrivial infinite-state systems: Lamport's bakery algorithm and Dijkstra's mutual exclusion algorithm for an unbounded number of processes (Section 5).

Related Work. Logical model checking is complementary to other infinite-state model checking techniques that symbolically represent a system's state space, such as regular languages [1], string/multiset grammars [4, 34], tree automata [22, 32], constraint logic programming [11], Presbuger arithmetic [6], program specialization [20], etc. Similar to logical model checking, they are often combined with abstraction methods, e.g., [5, 7, 21].

Our abstract logical model checking differs from these approaches in several ways. First, we do not impose restrictions in the formalisms used for the verification of properties. Except for requiring that equations have the finite variant property, the only condition imposed on the rewrite theories is being *topmost*, which is easily satisfied by many systems, including concurrent object-oriented systems. Second, the combination of different abstraction techniques can give a *faithful* abstraction that has *no* spurious counterexamples.

Similar to folding abstractions, there exist many infinite-state model checking methods to exploit an order relation $\preccurlyeq$, e.g., [14, 19]. However, those methods typically assume that $\preccurlyeq$ is well quasi-ordered (which implies well-foundedness of $\preccurlyeq$), while we do not impose such conditions on $\preccurlyeq$. Indeed, the E-subsumption relation $\preccurlyeq_E$ is, in general, *not* well-founded.

Bisimilar equational abstractions are also related to other abstraction techniques, e.g., [8, 25]. For rewrite theories, it is related to, and complements, abstraction techniques for rewrite theories such as [18, 30]. The main difference is that usual abstraction techniques do not provide bisimulations between the abstract and concrete systems and when they do provide them, they rely on manual proofs, instead than on simple, checkable criteria (such as those in Theorem 17) for defining bisimilar equational abstractions.

2 Preliminaries on Narrowing-based Logical Model Checking

An order-sorted signature is a triple $\Sigma = (S, \leq, \Sigma)$ with poset of sorts $(S, \leq)$ and operators Σ typed in $(S, \leq)$. The set $\mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ denotes the set of Σ-terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ denotes the set of ground Σ-terms of sort s. *Positions* in a term t are denoted as strings of nonzero natural numbers that represent tree positions when t is parsed as a tree. A subterm of a term t at a position p is denoted by $t|_p$, and the replacement in t of such a subterm by another term u is denoted by $t[u]_p$. A *substitution* $\sigma : Y \to \mathcal{T}_\Sigma(\mathcal{X})$ is a function from $Y \subseteq \mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$ such that σy has the same sort as that of $y \in Y$. The substitution instance σt is a term obtained from t by *simultaneously* replacing each occurrence of variable $y \in Dom(\sigma)$ in t with σy.

A Σ-*equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $s \in \Sigma$. Given a set E of Σ-equations, equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ [29]. The E-*subsumption* preorder $t \preccurlyeq_E t'$ holds iff there exists a substitution $\sigma : Y \to \mathcal{T}_\Sigma(\mathcal{X})$ such that $t =_E \sigma t'$, meaning that t' is *more general* than t modulo E. The E-renaming equivalence $t \approx_E t'$ holds iff there exists a substitution $\theta : \mathcal{X} \to \mathcal{X}$ such that $t =_E \theta t'$ and $\theta(x) \neq \theta(y)$ for any $x, y \in \mathcal{X}$, implying that $t \preccurlyeq_E t'$ and $t' \preccurlyeq_E t$.

An 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, written $l \longrightarrow r$, where l, r are Σ-terms. Each rule $l \longrightarrow r$ specifies a *one-step rewrite* $t \longrightarrow_{R,E} t'$ iff there is a non-variable position p in t and a substitution σ such that $t|_p =_E \sigma l$ and $t' = t[\sigma r]_p$. A rewrite theory $\mathcal{R}$ specifies a concurrent system whose states are axiomatized as the initial algebra $\mathcal{T}_{\Sigma/E}$ (i.e., each state is an E-equivalence class $[t]_E \in \mathcal{T}_{\Sigma/E}$ of ground terms), and whose concurrent transitions are axiomatized as one-step rewrites $\longrightarrow_{R,E}$ [28]. A rewrite theory $\mathcal{R}$ is *topmost* iff for each $l \longrightarrow r \in R$, $l, r \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}$ for a sort State at the top of one of the connected component of $(S, \leq)$, and no operator in Σ has State or any of its subsorts as an argument sort. This ensures that all rewrites with rules in R must take place at the top of the term.

For a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, state propositions can be defined by means of its equations. Each state proposition is defined as a term of sort Prop using (possibly parametric) function symbols of the form $p : \mathsf{s}_1 \dots \mathsf{s}_n \to \mathsf{Prop}$, and the satisfaction relation is defined by equations using the auxiliary operator $\_\models\_ : \mathsf{State}\ \mathsf{Prop} \to \mathsf{Bool}$, where sort Bool has two constants *true* and *false* such that $\mathit{true} \neq_E \mathit{false}$ and for any term $t \in \mathcal{T}_{\Sigma,\mathsf{Bool}}$ of sort Bool, either $t =_E \mathit{true}$ or $t =_E \mathit{false}$ holds. By definition, a state proposition $p(u_1, \dots, u_n) \in \mathcal{T}_{\Sigma/E,\mathsf{Prop}}$ is satisfied on $[t]_E \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$ iff $(t \models p(u_1, \dots, u_n)) =_E \mathit{true}$.

If $\mathcal{R}$ includes a set AP of state propositions *whose values on states are fully defined by its equations* E, we can associate to $\mathcal{R}$ a corresponding Kripke structure $\mathcal{K}(\mathcal{R})_{AP}$ for LTL model checking. A *Kripke structure* is a 4-tuple $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow \mathcal{K})$ with S a set of *states*, AP a set of *atomic state propositions*, $\mathcal{L} : S \to \mathcal{P}(AP)$ a *state-labeling function*, and $\longrightarrow_{\mathcal{K}} \subseteq S \times S$ a *total transition relation* where every state $s \in S$ has a next state $s' \in S$ with $s \longrightarrow_{\mathcal{K}} s'$. Given a subset $S_0 \subseteq S$, the set of its *successors* is $Post_{\mathcal{K}}(S_0) = \{s \in S \mid (\exists s_0 \in S_0)\ s_0 \longrightarrow_{\mathcal{K}} s\}$, and the set of its *reachable states* is $Post^*_{\mathcal{K}}(S_0) = \bigcup_{i \in \mathbb{N}} (Post_{\mathcal{K}})^i(S_0)$. We assume $\mathcal{R}$ is deadlock-free, since $\mathcal{R}$ can be transformed into an equivalent deadlock-free theory [30].

▶ **Definition 1.** Given $\mathcal{R} = (\Sigma, E, R)$ and a set AP of state propositions defined by its equations E, the corresponding Kripke structure is $\mathcal{K}(\mathcal{R})_{AP} = (\mathcal{T}_{\Sigma/E,\mathsf{State}}, AP, \mathcal{L}, \longrightarrow_{R,E})$, where $\mathcal{L}([t]_E) = \{p \in AP \mid (t \models p) =_E \mathit{true}\}$.

We present a topmost rewrite theory $\mathcal{R}$ specifying Lamport's bakery algorithm for mutual exclusion. Each state has the form "i ; j ; $[m_1] \dots [m_n]$," where i is the current number in the bakery's number dispenser, j is the number currently being served, and the $[m_1] \dots [m_n]$ are a multiset of customer processes, each in a *mode* m_l, which can be either *idle* (has not yet picked a number), or *wait*(n) (waiting with number n), or *crit*(n) (being served with number n). We model natural numbers as the free commutative monoid generated by 1 (denoted s) with multiset union (addition), denoted `__` (empty syntax), satisfying *associativity*, *commutativity*, and *identity* (`0`) axioms. For example, $0 = \mathsf{0}$, and $3 = s\,s\,s$. The behavior of the bakery algorithm is then specified by the following *topmost* rewrite rules:

```
rl [wake]: N ; M ; [idle] PS     =>  (s N) ; M ; [wait(N)] PS .
rl [crit]: N ; M ; [wait(M)] PS  =>  N ; M ; [crit(M)] PS .
rl [exit]: N ; M ; [crit(M)] PS  =>  N ; (s M) ; [idle] PS .
```

The state proposition `ex?` for the mutual exclusion is defined by the following equations, where the variable `WS` stands for a set of processes whose status is either *idle* or *wait*(n):

```
eq N ; M ; WS |= ex? = true .
eq N ; M ; [crit(M1)] WS |= ex? = true .
eq N ; M ; [crit(M1)] [crit(M2)] PS |= ex? = false .
```

This system is infinite-state *in two ways*: (i) the counters i and j are unbounded; and (ii) the number n of customer processes is also unbounded. For example, given the initial state "`0 ; 0 ; [idle]`," we obtain the infinite transition system of Figure 1.

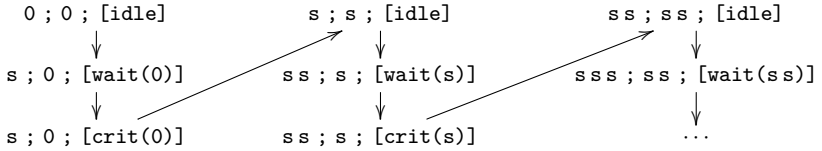


Figure 1 An infinite transition system for the Bakery algorithm from "`0 ; 0 ; [idle]`."

Narrowing [23, 24] generalizes term rewriting by allowing free variables in terms and by performing unification instead of matching. An *E-unifier* for an equation $t = t'$ is a substitution σ such that $\sigma t =_E \sigma t'$, and a set $CSU_E(t = t')$ of E-unifiers is *complete* iff any E-unifier ρ for $t = t'$ has a more general substitution σ in $CSU_E(t = t')$, i.e., there is a substitution η such that $\rho =_E \sigma \circ \eta$. Given a *topmost* rewrite theory $\mathcal{R} = (\Sigma, E, R)$, if the left-hand sides of the rules R are non-variable terms and a finitary E-unification procedure is available, each rule $l \longrightarrow r$ specifies a *topmost narrowing step* $t \rightsquigarrow_{\sigma,R,E} t'$ (or $t \rightsquigarrow_{R,E} t'$) iff there exists an E-unifier $\sigma \in CSU_E(t = l)$ such that $t' = \sigma r$.

If an equational theory (Σ, E) has the *finite variant property*, there is an algorithm to compute a finitary and complete set $CSU_E(t = t')$ of E-unifiers [17]. An E-variant of a term t is a pair (t', θ) with t' an E-canonical form of a substitution instance θt, i.e., $\theta t \longrightarrow_E^* t'$ and t' cannot be further rewritten. A variant (t_2, θ_2) is more general than (t_1, θ_1) iff there is a substitution η such that $t_1 =_E \eta t_2$ and $\theta_1 =_E \theta_2 \circ \eta$. An equational theory (Σ, E) has the finite variant property iff the set of most general E-variants for each term is finite (see [10, 17] for details). For the Bakery example, the equations for the state proposition `ex?` trivially satisfy the finite variant property because their right-hand sides are all constants.

Such a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ also specifies a *logical* transition system $\mathcal{N}_\mathcal{R}$ [16] whose states are elements of the algebra $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ of sort State (excluding variables as states), and whose transitions are specified by topmost narrowing steps $\rightsquigarrow_{R,E}$. That is, the states of $\mathcal{N}_\mathcal{R}$ are not *concrete states* (i.e., ground terms), but *state patterns*, that is, terms $t(x_1, \ldots, x_n)$ with *logical variables* $x_1, \ldots, x_n$. What $t(x_1, \ldots, x_n)$ stands for is *not* a single state, but the set of all concrete states $[\theta t] \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$ that are its *ground instances*.

If a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ defines a *finite* set AP of state propositions by its equations E, we can also associate to $\mathcal{R}$ a corresponding *narrowing-based logical Kripke structure* $\mathcal{N}_\mathcal{R}^{AP}$. Each state in the underlying logical transition system $\mathcal{N}_\mathcal{R}$ is now split into possibly several states in $\mathcal{N}_\mathcal{R}^{AP}$ (by $\rightsquigarrow_{AP}$ in the following definition) so that the truth of every state proposition for each state in $\mathcal{N}_\mathcal{R}^{AP}$ is decided into *true* or *false*.

▶ **Definition 2.** [16] Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a finite set $AP = \{p_1, \ldots, p_n\}$ of state propositions defined by E, its narrowing-based *logical Kripke structure* is:

$$\mathcal{N}_\mathcal{R}^{AP} = \big(N_\mathcal{R}^{AP},\ AP,\ \mathcal{L},\ (\rightsquigarrow_{R,E};\rightsquigarrow_{AP})\big)$$

where $t\ (\rightsquigarrow_{R,E};\rightsquigarrow_{AP})\ t' \iff (\exists u)\ t \rightsquigarrow_{R,E} u \rightsquigarrow_{AP} t'$, and

- $N_\mathcal{R}^{AP} = \{[t]_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\mathsf{State}} - \mathcal{X} \mid (\forall p \in AP)\ (t \models p) =_E \mathit{true} \vee (t \models p) =_E \mathit{false}\}$,
- $\mathcal{L}([t]_E) = \{p \in AP \mid (t \models p) =_E \mathit{true}\}$,
- $t \rightsquigarrow_{AP} t' \iff \exists \theta \in CSU_E\big((t \models p_1) = w_1 \wedge \cdots \wedge (t \models p_n) = w_n\big)$ such that $t' = \theta t$, where for each $1 \leq i \leq n$, w_i is either *true* or *false*.

For the Bakery example, given the *logical* initial state `N ; N ; [`*idle*`]`, we obtain within $\mathcal{N}_\mathcal{R}$ the infinite transition system in Figure 2. The corresponding Kripke structure $\mathcal{N}_\mathcal{R}^{\{\texttt{ex?}\}}$ is then similar to $\mathcal{N}_\mathcal{R}$, but each logical state in $\mathcal{N}_\mathcal{R}$ is split according to the truth of `ex?`.

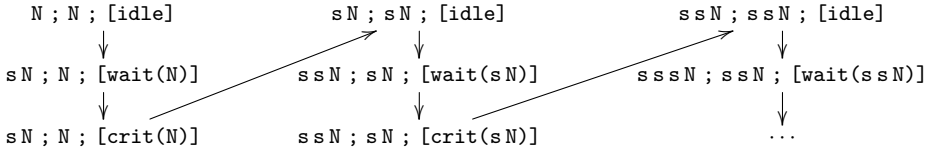


Figure 2 An infinite transition system with *logical* states for the Bakery algorithm.

Such a narrowing-based Kripke structure $\mathcal{N}_{\mathcal{R}}^{AP}$ can be considered as an *exact abstraction* of the concrete Kripke structure $\mathcal{K}(\mathcal{R})_{AP}$, where concrete states are abstracted by means of the *E-subsumption* preorder $\preccurlyeq_E$ as shown in the following theorem.

▶ **Theorem 3.** *[16] Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a finite set AP of state propositions defined by E, for an LTL formula φ and a pattern $t \in N_{\mathcal{R}}^{AP}$:*

$$\mathcal{N}_{\mathcal{R}}^{AP}, [t]_E \models \varphi \iff (\forall \theta : \mathcal{X} \to \mathcal{T}_\Sigma)\ \mathcal{K}(\mathcal{R})_{AP}, [\theta t]_E \models \varphi.$$

However, $\mathcal{N}_{\mathcal{R}}^{AP}$ often has an infinite number of *logical* states as in Figure 2. The following sections explain how we can reduce it to a finite state space by abstraction techniques that provide either an over-approximation or an under-approximation of $\mathcal{N}_{\mathcal{R}}^{AP}$.

3 Abstract Logical Model Checking

For model checking techniques, an abstraction $\widehat{\mathcal{K}}$ of a concurrent system typically preserves all moves of the original system $\mathcal{K}$, in terms of a *simulation* between $\mathcal{K}$ and $\widehat{\mathcal{K}}$. Given two Kripke structures $\mathcal{K}_i = (S_i, AP, \mathcal{L}_i, \longrightarrow_{\mathcal{K}_i})$, $i = 1, 2$, a binary relation $H \subseteq S_1 \times S_2$ is a *simulation* iff (i) $s_1 H s_2$ and $s_1 \longrightarrow_{\mathcal{K}_1} s_1'$ implies that $(\exists s_2' \in S_2)$ $s_1' H s_2'$ and $s_2 \longrightarrow_{\mathcal{K}_2} s_2'$, and (ii) $s_1 H s_2 \implies \mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$. A simulation $H \subseteq S_1 \times S_2$ is *total* iff for any $s_1 \in S_1$ there exists $s_2 \in S_2$ such that $s_1 H s_2$. H is a *bisimulation* iff both H and H^{-1} are simulations. If $\mathcal{K}_2$ *simulates* $\mathcal{K}_1$, any LTL formula satisfied in $\mathcal{K}_2$ is also satisfied in $\mathcal{K}_1$.

▶ **Lemma 4.** *[8] Given $\mathcal{K}_i = (S_i, AP, \mathcal{L}_i, \longrightarrow_{\mathcal{K}_i})$, $i = 1, 2$, for a simulation $H \subseteq S_1 \times S_2$, if $s_0^1 H s_0^2$, then for any LTL formula φ over AP, $\mathcal{K}_2, s_0^2 \models \varphi$ implies $\mathcal{K}_1, s_0^1 \models \varphi$.*

This section presents two abstraction techniques for narrowing-based logical model checking, namely, *folding abstractions* and *equational abstractions*. Such an abstraction $\widehat{\mathcal{K}}$ provides an over-approximation of the original system $\mathcal{K}$, i.e., $\widehat{\mathcal{K}}$ *simulates* $\mathcal{K}$. Thus, if we verify that φ holds for $\widehat{\mathcal{K}}$, then we can be sure that it also holds for $\mathcal{K}$. However, as usual for over-approximation abstraction techniques, a counterexample in $\widehat{\mathcal{K}}$ can be *spurious*, so that it has no counterpart in $\mathcal{K}$. We also provide some conditions for both abstraction techniques when $\widehat{\mathcal{K}}$ can be *faithful* for a certain *subset* Δ of LTL formulas so that $\widehat{\mathcal{K}}$ generates *no* spurious counterexamples, i.e., for each $\varphi \in \Delta$ and $s\, H\, \hat{s}$, $\widehat{\mathcal{K}}, \hat{s} \models \varphi$ iff $\mathcal{K}, s \models \varphi$.

3.1 Folding Abstractions

We can reduce a logical Kripke structure by collapsing each state into a more *general* state according to the E-subsumption preorder $\preccurlyeq_E$, by the notion of folding abstraction proposed in [16]. In this paper we further generalize folding abstractions with any *folding preorder* $\preccurlyeq$.

▶ **Definition 5.** Given a Kripke structure $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$, a *folding preorder* $\preccurlyeq \subseteq S^2$ is a reflexive and transitive relation on S that defines a simulation between $\mathcal{K}$ and $\mathcal{K}$.

For a narrowing-based Kripke structure $\mathcal{N}_{\mathcal{R}}^{AP}$, the most common folding relations are: equality modulo E, renaming modulo E, and matching modulo E, i.e., $\preccurlyeq \in \{=_E, \approx_E, \preccurlyeq_E\}$ [16].

We can iteratively construct a *folding abstraction* of a Kripke structure $\mathcal{K}$ from a set of initial states $I \subseteq S$, using a folding preorder $\preccurlyeq \subseteq S^2$ as shown in Definition 6 below. Each state $s \in S$ in $\mathcal{K}$ is collapsed into a *previously seen* state $t \in S$ such that $s \preccurlyeq t$, while any transition for the folded state s is transferred to the state t in the folding abstraction. Such a folded Kripke structure has in general much fewer states than the original structure, and can sometimes collapse an infinite-state space to a finite-state one.

▶ **Definition 6** (Folding Abstraction). Given $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, the *folding abstraction* of $\mathcal{K}$ from I is the Kripke structure

$$\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I) = (Post^{*}_{\mathcal{K}\preccurlyeq}(I),\ AP,\ \mathcal{L},\ \longrightarrow_{\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I)})$$

where $Post^{*}_{\mathcal{K}\preccurlyeq}(I) = \bigcup_{i\in\mathbb{N}} Post^{i}_{\mathcal{K}\preccurlyeq}(I)$ and $\longrightarrow_{\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I)} = \bigcup_{i\in\mathbb{N}} \longrightarrow^{\preccurlyeq}_{\mathcal{K},i}$ such that:

- $Post^{n+1}_{\mathcal{K}\preccurlyeq}(I)$ is the successor set of $Post^{n}_{\mathcal{K}\preccurlyeq}(I)$ *not* subsumed by previously seen states:

$$Post^{0}_{\mathcal{K}\preccurlyeq}(I) = I, \quad Post^{n+1}_{\mathcal{K}\preccurlyeq}(I) = \{s \in Post_{\mathcal{K}}(Post^{n}_{\mathcal{K}\preccurlyeq}(I)) \mid \forall l \le n\ \forall u \in Post^{l}_{\mathcal{K}\preccurlyeq}(I).\ s \not\preccurlyeq u\}.$$

- $\longrightarrow^{\preccurlyeq}_{\mathcal{K},n+1} \subseteq Post^{n}_{\mathcal{K}\preccurlyeq}(I) \times \left[\bigcup_{0\le i\le n+1} Post^{i}_{\mathcal{K}\preccurlyeq}(I)\right]$ defines the transitions from each state $s \in Post^{n}_{\mathcal{K}\preccurlyeq}(I)$ to a next state $t \in Post^{l}_{\mathcal{K}\preccurlyeq}(I)$ for $0 \le l \le n+1$, up to $n+1$ steps:

$$\longrightarrow^{\preccurlyeq}_{\mathcal{K},0} = \emptyset, \qquad s \longrightarrow^{\preccurlyeq}_{\mathcal{K},n+1} s' \iff \exists t \in Post_{\mathcal{K}}(s).\ t \preccurlyeq s'.$$

Each reachable state $s \in Post^{*}_{\mathcal{K}}(I)$ in a Kripke structure $\mathcal{K}$ has a corresponding abstract state $\hat{s} \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$ in the folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I)$ as follows.

▶ **Lemma 7.** *Given $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow \mathcal{K})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, for each reachable state $s \in Post^{*}_{\mathcal{K}}(I)$, there is $\hat{s} \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$ such that $s \preccurlyeq \hat{s}$.*

Proof. For a reachable state $s \in Post^{*}_{\mathcal{K}}(I)$ of $\mathcal{K}$, there exists a finite path $\pi_s : [n] \to S$ with length $n \in \mathbb{N}$ beginning in I and ending at s (i.e., $\pi_s(0) \in I$ and $\pi_s(n-1) = s$), where $[n] = \{0, 1, \ldots, n\}$. We show this lemma by induction on the length of π_s. First, if $|\pi_s| = 0$, then $s \in I = Post^{0}_{\mathcal{K}\preccurlyeq}(I) \subseteq Post^{*}_{\mathcal{K}\preccurlyeq}(I)$, and $s \preccurlyeq s$. Next, suppose that for any path π beginning in I with length n, there exists an abstract state $t_{n-1} \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$ such that $\pi(n-1) \preccurlyeq t_{n-1}$. Consider a path π_s with length $n+1$ such that $\pi_s(0) \in I$ and $\pi_s(n) = s$. By induction hypothesis, there exists $t_{n-1} \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$ such that $\pi_s(n-1) \preccurlyeq t_{n-1}$. Notice that $t_{n-1} \in Post^{k}_{\mathcal{K}\preccurlyeq}(I)$ for some $k \in \mathbb{N}$. Since $\preccurlyeq$ is a simulation between $\mathcal{K}$ and $\mathcal{K}$:

$$\begin{array}{ccccccc} \pi_s(n-1) & \in & Post^{*}_{\mathcal{K}}(I) & \longrightarrow_{\mathcal{K}} & \pi_s(n) & \in & Post^{*}_{\mathcal{K}}(I) \\ \preccurlyeq & & & & \preccurlyeq & & \\ t_{n-1} & \in & Post^{k}_{\mathcal{K}\preccurlyeq}(I) & \longrightarrow_{\mathcal{K}} & \exists\, t_n & \in & Post_{\mathcal{K}}(Post^{k}_{\mathcal{K}\preccurlyeq}(I)) \end{array}$$

There are now two possibilities: (i) if $t_n \in Post^{k+1}_{\mathcal{K}\preccurlyeq}(I)$, we found $t_n \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$ such that $s = \pi_s(n) \preccurlyeq t_n$; (ii) otherwise, there exist $l \le k$ and $u \in Post^{l}_{\mathcal{K}\preccurlyeq}(I)$ such that $t_n \preccurlyeq u$, since by definition $Post^{k+1}_{\mathcal{K}\preccurlyeq}(I) = \{s \in Post_{\mathcal{K}}(Post^{k}_{\mathcal{K}\preccurlyeq}(I)) \mid \forall l \le k\ \forall u \in Post^{l}_{\mathcal{K}\preccurlyeq}(I).\ s \not\preccurlyeq u\}$; that is, we found $u \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$ such that $s = \pi_s(n) \preccurlyeq t_n \preccurlyeq u$. ◀

Furthermore, the folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I)$ of $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$ simulates the *reachable* substructure $\mathcal{R}each_{\mathcal{K}}(I)$ of $\mathcal{K}$ that only contains reachable states from I, where $\mathcal{R}each_{\mathcal{K}}(I) = (Post^{*}_{\mathcal{K}}(I),\ AP,\ \mathcal{L},\ \longrightarrow_{\mathcal{K}} \cap\, (Post^{*}_{\mathcal{K}}(I))^2)$.

▶ **Theorem 8.** *Given $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow \mathcal{K})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, the folding preorder $\preccurlyeq$ is a total simulation between $\mathcal{R}each_{\mathcal{K}}(I)$ and $\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I)$.*

Proof. Suppose $s \preccurlyeq t$ and $s \longrightarrow_{\mathcal{K}} s'$ for states $s, s' \in Post^{*}_{\mathcal{K}}(I)$ and an abstract state $t \in Post^{*}_{\mathcal{K}\preccurlyeq}(I)$. By definition, $t \in Post^{k}_{\mathcal{K}\preccurlyeq}(I)$ for some $k \in \mathbb{N}$. Since $\preccurlyeq$ is a simulation between $\mathcal{K}$ and $\mathcal{K}$, there exists $t' \in Post_{\mathcal{K}}(Post^{k}_{\mathcal{K}\preccurlyeq}(I))$ such that $t \longrightarrow_{\mathcal{K}} t'$ and $s' \preccurlyeq t'$. There are also two possibilities: (i) if $t' \in Post^{k+1}_{\mathcal{K}\preccurlyeq}(I)$, we have $t \longrightarrow^{\preccurlyeq}_{\mathcal{K},k+1} t'$ such that $s' \preccurlyeq t'$; (ii) otherwise, by definition of $Post^{k+1}_{\mathcal{K}\preccurlyeq}(I)$, there exist $l \le k$ and $t'' \in Post^{l}_{\mathcal{K}\preccurlyeq}(I)$ such that $t' \preccurlyeq t''$, and we have $t \longrightarrow^{\preccurlyeq}_{\mathcal{K},k+1} t''$ again, where $s' \preccurlyeq t' \preccurlyeq t''$. Therefore, $\preccurlyeq$ is a simulation between $\mathcal{R}each_{\mathcal{K}}(I)$ and $\mathcal{R}each^{\preccurlyeq}_{\mathcal{K}}(I)$. Also, $\preccurlyeq$ is total by Lemma 7. ◀

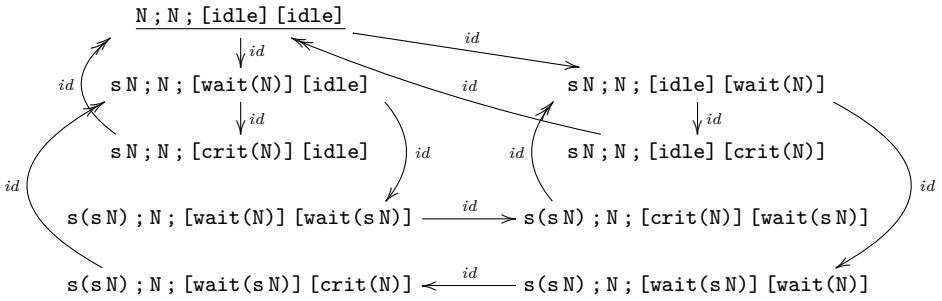


Figure 3 A finite folding abstraction with a folding preorder $\preccurlyeq_E$ for the Bakery algorithm. We add a double-headed arrow between states A and C to denote both a transition from state A to another state B and the fact that state B is folded into state C.

For our Bakery example, given the *logical* initial state N ; N ; [*idle*] [*idle*], Figure 3 shows the *finite* folding abstraction of $\mathcal{N}_{\mathcal{R}}^{\{\texttt{ex?}\}}$ with the folding preorder $\preccurlyeq_E$. The mutual exclusion $\Box$ex? is satisfied in the folding abstraction, since the state proposition ex? evaluates to *true* in every state. Thanks to Theorem 8, $\Box$ex? is satisfied for any possible instance of it.

3.2 Faithfulness of Folding Abstractions

A folding abstraction $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$ is in general an over-approximation of a logical state space. If an LTL formula φ is *not* satisfied in $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$, it can generate a spurious counterexample for φ. Nonetheless, if a folding preorder $\preccurlyeq$ is *symmetric*, then $\preccurlyeq$ becomes a total bisimulation by Theorem 8, so that both satisfy exactly the same set of LTL formulas. For example, both $=_E$ and $\approx_E$ are symmetric for a narrowing-based Kripke structure.

What can we then say about $\preccurlyeq_E$ for a narrowing-based Kripke structure? Since $\preccurlyeq_E$ is more general than $=_E$ and $\approx_E$, it has a better chance to yield a finite state space, although it may generate a spurious counterexample for an LTL formula. However, a folding abstraction is *faithful* for invariants; that is, if there is a counterexample for any invariant $\Box\Phi$ in $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$, where Φ is a boolean formula with no temporal operators, there exists a *real* counterexample in $\mathcal{K}$. This faithfulness follows from the fact that each state in $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$ is still *reachable* from I in the original Kripke structure $\mathcal{K}$.

▶ **Lemma 9.** *Given a Kripke structure $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, we have $Post^*_{\mathcal{K}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{K}}(I)$.*

Proof. Recall that $Post^*_{\mathcal{K}_{\preccurlyeq}}(I) = \bigcup_{i\in\mathbb{N}} Post^i_{\mathcal{K}_{\preccurlyeq}}(I)$. By definition, $Post^0_{\mathcal{K}_{\preccurlyeq}}(I) = I \subseteq Post^*_{\mathcal{K}}(I)$. Suppose that $Post^n_{\mathcal{K}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{K}}(I)$ for some $n \in \mathbb{N}$. Since $Post^{n+1}_{\mathcal{K}_{\preccurlyeq}}(I) \subseteq Post_{\mathcal{K}}(Post^n_{\mathcal{K}_{\preccurlyeq}}(I))$, for each $s' \in Post^{n+1}_{\mathcal{K}_{\preccurlyeq}}(I)$, there exists $s \in Post^n_{\mathcal{K}_{\preccurlyeq}}(I)$ such that $s \longrightarrow_{\mathcal{K}} s'$. By induction hypothesis, $s \in Post^*_{\mathcal{K}}(I)$, and thus $s' \in Post^*_{\mathcal{K}}(I)$. Therefore, $Post^{n+1}_{\mathcal{K}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{K}}(I)$. ◀

If a folding abstraction $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$ does *not* satisfy an invariant, then there exists an *error* state $s \in Post^*_{\mathcal{K}_{\preccurlyeq}}(I)$ in $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$ that violates the invariant. Because the error state s is again reachable from I in the original Kripke structure $\mathcal{K}$, we can construct a *concrete counterexample* in $\mathcal{K}$ by backward search from s to I. Consequently:

▶ **Theorem 10** (Faithfulness for Invariants). *Given a Kripke structure $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow \mathcal{K})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, for any invariant $\Box\Phi$:*

$$\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I), I \models \Box\Phi \iff \mathcal{K}, I \models \Box\Phi.$$

Moreover, folding abstractions can provide a faithful model checking procedure for *safety* LTL formulas. For a safety LTL formula φ, there exists a *finite automaton* $\mathcal{F}_{\neg\varphi}$ that recognizes counterexamples for φ [2, 26]. A finite automaton is a 5-tuple $\mathcal{F} = (Q, Q_0, \mathcal{P}(AP), \delta, F)$ with Q a finite set of states, $Q_0 \subseteq Q$ a set of initial states, $\mathcal{P}(AP)$ an alphabet of transition labels, $\delta \subseteq Q \times \mathcal{P}(AP) \times Q$ a transition relation, and $F \subseteq Q$ a set of final states. The *language* accepted by $\mathcal{F}$ is the set $L(\mathcal{F})$ of finite runs of $\mathcal{F}$ starting in Q_0 and ending in F. Given a Kripke structure $\mathcal{K}$ and a set $I \subseteq S$ of initial states, the *synchronous product* of $\mathcal{K}$ and $\mathcal{F}$ is a finite automaton $\mathcal{K}[I] \times \mathcal{F} = (S \times Q, I \times Q_0, \mathcal{P}(AP), \delta_{\mathcal{K}}, S \times F)$ such that $(s,b) \xrightarrow{\mathcal{L}(s)} (s',b') \in \delta_{\mathcal{K}}$ iff $s \longrightarrow_{\mathcal{K}} s' \wedge b \xrightarrow{\mathcal{L}(s)} b' \in \delta$. The model checking problem of a *safety* LTL formula φ can then be characterized by using a finite automaton $\mathcal{F}_{\neg\varphi}$ associated to the negated formula $\neg\varphi$, where $\mathcal{K}, I \models \varphi$ iff $L(\mathcal{K}[I] \times \mathcal{F}_{\neg\varphi}) = \emptyset$ [2, 26].

Since the emptiness checking of the finite automaton $\mathcal{K}[I] \times \mathcal{F}_{\neg\varphi}$ can be characterized by the reachability analysis of the final states, we can apply our previous result to *faithfully* abstract the synchronous product $\mathcal{K}[I] \times \mathcal{F}_{\neg\varphi}$. For a folding preorder $\preccurlyeq$ of $\mathcal{K}$, let the *product preorder* $\preccurlyeq_{\mathcal{F}} \subseteq (S \times Q)^2$ be defined by the equivalence: $(s,b) \preccurlyeq_{\mathcal{F}} (s',b') \iff s \preccurlyeq s' \wedge b = b'$.

▶ **Lemma 11.** *Given a finite automaton $\mathcal{F}$ and a folding preorder $\preccurlyeq$ for a Kripke structure $\mathcal{K}$, the product preorder $\preccurlyeq_{\mathcal{F}}$ is a folding preorder for the synchronous product $\mathcal{K}[I] \times \mathcal{F}$.*

Proof. Suppose that $(s_1,b_1) \xrightarrow{\mathcal{L}(s_1)} (s_1',b_1') \in \delta_{\mathcal{K}}$ and $(s_1,b_1) \preccurlyeq_{\mathcal{F}} (s_2,b_2)$. By definition, $s_1 \preccurlyeq s_2$ and $b_1 = b_2$. Since $\preccurlyeq$ is a simulation between $\mathcal{K}$ and $\mathcal{K}$, there exists $s_2' \in S$ such that $s_2 \longrightarrow_{\mathcal{K}} s_2'$ and $s_1' \preccurlyeq s_2'$, and $\mathcal{L}(s_1) = \mathcal{L}(s_2)$. Hence, $(s_2,b_2) \xrightarrow{\mathcal{L}(s_1)} (s_2',b_1') \in \delta_{\mathcal{K}}$, and $(s_1',b_1') \preccurlyeq_{\mathcal{F}} (s_2',b_1')$. Therefore, $\preccurlyeq_{\mathcal{F}}$ is a simulation between $\mathcal{K}[I] \times \mathcal{F}$ and $\mathcal{K}[I] \times \mathcal{F}$. ◀

Therefore, by Theorem 10, for a safety LTL formula φ, $L\big(\mathcal{R}each^{\preccurlyeq_{\mathcal{F}_{\neg\varphi}}}_{\mathcal{K}[I]\times\mathcal{F}_{\neg\varphi}}(I \times Q_0)\big) = \emptyset$ iff $L(\mathcal{K}[I] \times \mathcal{F}_{\neg\varphi}) = \emptyset$. Consequently, we have:

▶ **Theorem 12** (Faithfulness for Safety Properties). *Given $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow \mathcal{K})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, for a* safety *LTL formula φ, there exists a* finite *automaton $\mathcal{F}_{\neg\varphi}$ with Q_0 a set of initial states such that:*

$$L\big(\mathcal{R}each^{\preccurlyeq_{\mathcal{F}_{\neg\varphi}}}_{\mathcal{K}[I]\times\mathcal{F}_{\neg\varphi}}(I \times Q_0)\big) = \emptyset \iff \mathcal{K}, I \models \varphi.$$

3.3 Equational Abstractions for Logical State Space

A logical state representation for a rewrite theory $\mathcal{R}$ already affords a huge abstraction, and it may turn an infinite system into a more manageable system. However, even a folding abstraction of such a logical state space need not be finite in general. For example, when we consider our Bakery example and the logical initial state `N ; N ; IS` that does not bound the number of customer processes, the folding abstraction with $\preccurlyeq_E$ has an infinite path that keeps incrementing the number of processes with instantiation, as shown in Figure 4.

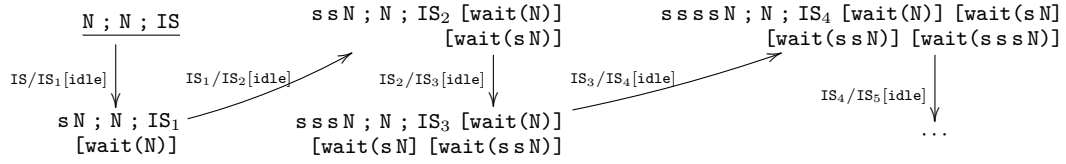


Figure 4 An infinite folded logical transition system for the Bakery algorithm with an arbitrary number of processes. The logical variable $\mathtt{IS}_k$ stands for a set of `idle` processes.

An abstraction of a concurrent system can be constructed by a suitable equivalence relation $\equiv$ on states [8, 30]. Given $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$ and an equivalence relation $\equiv \subseteq S \times S$ such that $s_1 \equiv s_2$ implies $\mathcal{L}(s_1) = \mathcal{L}(s_2)$, the *quotient abstraction* $\mathcal{K}/\equiv$ is a Kripke structure $(S/\equiv, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}/\equiv})$ where $[s_1] \longrightarrow_{\mathcal{K}/\equiv} [s_2]$ iff $(\exists s_1' \in [s_1], s_2' \in [s_2])\ s_1' \longrightarrow_{\mathcal{K}} s_2'$. For a topmost and deadlock-free rewrite theory $\mathcal{R} = (\Sigma, E, R)$ containing a set AP of state propositions defined by the equations E, by adding a set of extra equations G to $\mathcal{R}$, we can define an *equational abstraction* $\mathcal{R}/G = (\Sigma, E \cup G, R)$ [30], which specifies the quotient abstraction of $\mathcal{K}(\mathcal{R})_{AP}$ by the equivalence relation $\equiv_G$ on states, namely, $[t]_E \equiv_G [t']_E \iff t =_{E \cup G} t'$, where $[t]_E \equiv_G [t']_E$ implies $\mathcal{L}([t]_E) = \mathcal{L}([t']_E)$.

In this section we explain how equational abstractions can be applied for narrowing-based model checking for collapsing an infinite *logical* state space into a finite one, whereas equation abstractions [30] are used for ground terms in the literature for non-logical state spaces.

▶ **Definition 13** (Equational Abstraction)**.** Given $\mathcal{R} = (\Sigma, E, R)$ and a set of equations G, the rewrite theory $\mathcal{R}/G = (\Sigma, E \cup G, R)$ defines an *equational abstraction* iff: (i) finitary unification procedures modulo E and modulo $E \cup G$ are available, and (ii) *true* $\neq_{E \cup G}$ *false*.

If a set AP of state propositions is fully defined by E, whenever $t =_{E \cup G} t'$ for two states $t, t' \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$, condition (ii) ensures that both t and t' satisfy exactly the same state propositions. Note that if $E \cup G$ has the finite variant property, there is a finitary unification procedure module $E \cup G$ as explained in [10, 17], which is available in the Maude system [12].

Similar to equational abstractions for a concrete Kripke structure $\mathcal{K}(\mathcal{R})_{AP}$, we obtain a simulation between a logical Kripke structure $\mathcal{N}_{\mathcal{R}}^{AP}$ and its equational abstraction $\mathcal{N}_{\mathcal{R}/G}^{AP}$.

▶ **Lemma 14.** *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a finite set AP of state propositions defined by E, and a set G of equations, if $\mathcal{R}/G$ is an equational abstraction, then $H_G = \{([t]_E, [t]_{E \cup G}) \mid t \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}\}$ is a simulation between $\mathcal{N}_{\mathcal{R}}^{AP}$ and $\mathcal{N}_{\mathcal{R}/G}^{AP}$.*

Proof. For $t \in \mathcal{T}_\Sigma(\mathcal{X})$ and $u \in \mathcal{T}_{\Sigma/G}(\mathcal{X})$, suppose $t =_{E \cup G} u$ and $t \rightsquigarrow_{\theta,R,E} t'$ using rule $l \longrightarrow r \in R$, that is, $\theta \in CSU_E(t = l)$ and $t' = \theta(r)$. Since $\theta \in CSU_E(t = l)$, there exists $\theta' \in CSU_{E \cup G}(u = l)$ such that $\theta =_{E \cup G} \theta'$. Therefore, for $u' = \theta' r$, $u \rightsquigarrow_{\theta',R,E \cup G} u'$ using the same rule $l \longrightarrow r \in R$ and $t' = \theta r =_{E \cup G} \theta' r = u'$. ◀

We introduce *bisimilar equational abstractions*, which ensure a bisimulation between the narrowing-based Kripke structure $\mathcal{N}_{\mathcal{R}}^{AP}$ and its quotient abstraction $\mathcal{N}_{\mathcal{R}/G}^{AP}$.

▶ **Definition 15** (Bisimilar Equational Abstraction)**.** Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a set G of extra equations, an equational abstraction $\mathcal{R}/G$ is a *bisimilar equational abstraction* iff for any states $t_1, t_2, t_3 \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$:

$$t_1 \longrightarrow_{R,E} t_2 \ \wedge\ t_1 =_{E \cup G} t_3 \implies (\exists t_4 \in \mathcal{T}_{\Sigma/E,\mathsf{State}})\ t_3 \longrightarrow_{R,E} t_4 \ \wedge\ t_2 =_{E \cup G} t_4.$$

Notice that for a bisimilar equational abstraction $\mathcal{R}/G$, the equivalence relation $=_{E \cup G}$ is indeed a bisimulation for $\mathcal{R}$ with respect to $\longrightarrow_{R,E}$, since $=_{E \cup G}$ is symmetric.

▶ **Theorem 16.** *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a finite set AP of state propositions defined by the equations E, if $\mathcal{R}/G$ is a bisimilar equational abstraction, then $H_G = \{([t]_E, [t]_{E \cup G}) \mid t \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}\}$ is a bisimulation between $\mathcal{N}_{\mathcal{R}}^{AP}$ and $\mathcal{N}_{\mathcal{R}/G}^{AP}$.*

Proof. We only need to prove the H_G is a simulation between $\mathcal{N}_{\mathcal{R}/G}^{AP}$ and $\mathcal{N}_{\mathcal{R}}^{AP}$. For terms $u \in \mathcal{T}_{\Sigma/G}(\mathcal{X})$ and $t \in \mathcal{T}_\Sigma(\mathcal{X})$, suppose $u =_{E \cup G} t$ and $u \rightsquigarrow_{\sigma,R,E \cup G} u'$, i.e., $\sigma u \longrightarrow_{R,E \cup G} u'$. Since $\sigma u =_{E \cup G} \sigma t$, by definition of bisimilar equational abstractions, there exists a term $t' \in \mathcal{T}_{\Sigma/E}$ such that $\sigma t \longrightarrow_{R,E} t'$ and $u' =_{E \cup G} t'$. Therefore, by completeness of narrowing, there exists a substitution σ' such that $t \rightsquigarrow_{\sigma',R,E} t'$. ◀

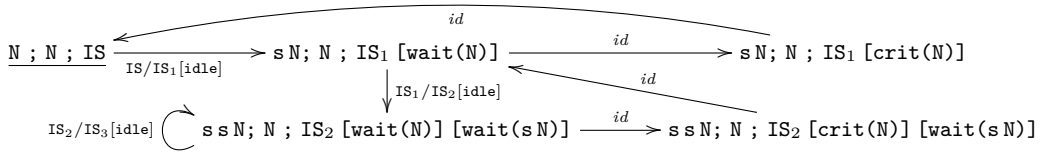


Figure 5 An abstract folded logical transition system for the bakery example.

A bisimilar equational abstraction with *topmost equations* of the form $t = t'$ with $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}$ can be easily identified by checking critical pairs between the left-hand sides of the rules and both sides of the equations in the equational abstraction, and checking that application of an equation does not interfere with the application of a rewrite rule. This process can easily be automated in a similar way to the existing Maude coherence checker [13], which checks similar (but slightly different) conditions between equations and rules.

▶ **Theorem 17** (Necessary/Sufficient Conditions for Bisimilarity). *Given a topmost rewrite theory $\mathcal{R}$ and an equational abstraction $\mathcal{R}/G$ for a set G of* topmost *equations, $\mathcal{R}/G$ is a bisimilar equational abstraction iff for each rule $l \longrightarrow r$ and each $u = v \in G$ or $v = u \in G$:*

$$\sigma \in CSU_E(l = u) \implies \big(\exists \theta : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})\big)\ \sigma v =_E \theta l \ \wedge\ \sigma r =_{E \cup G} \theta r. \qquad (*)$$

Proof. (If) Suppose that $t_1 \longrightarrow_{R,E} t_2$ and $t_1 =_{E \cup G} t_3$. If $=^k_{G/E}$ denotes k applications of equations in G modulo E, then $t_1 =^n_{G/E} t_3$ for some $n \in \mathbb{N}$. We prove by induction on the number n that $(\exists t_4)\ t_3 \longrightarrow_{R,E} t_4$ and $t_2 =_{E \cup G} t_4$. When $n = 0$, it is immediate because $t_1 =_E t_3$ and then $t_3 \longrightarrow_{R,E} t_2$. For $n > 0$, assume that if $t_1 =^n_{G/E} t'_3$ for any $t'_3 \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$, there exists $t'_4 \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$ such that $t'_3 \longrightarrow_{R,E} t'_4$ and $t_2 =_{E \cup G} t'_4$. If $t_1 =^n_{G/E} t'_3 =^1_{G/E} t_3$, then $t'_3 \longrightarrow_{R,E} t'_4$ and by using the critical pair condition in the statement, $(\exists t_4)\ t_3 \longrightarrow_{R,E} t_4$ and $t'_4 =_{E \cup G} t_4$. Finally, we have that $t_2 =_{E \cup G} t'_4 =_{E \cup G} t_4$ and the conclusion follows.
(Only if) The property $t_1 \longrightarrow_{R,E} t_2 \ \wedge\ t_1 =_{E \cup G} t_3 \implies (\exists t_4)\ t_3 \longrightarrow_{R,E} t_4 \ \wedge\ t_2 =_{E \cup G} t_4$ must be satisfied for a term t_3 that has only one application of the equations in G, i.e., $t_1 =^1_{G/E} t_3$. And such a case is indeed represented by the conditions of the statement. ◀

The reason why only topmost equations are allowed for bisimilar equational abstractions is to avoid problems caused by repeated variables in transition rules. For instance, consider $R = \{f(X, X) \longrightarrow h(X)\}$, $E = \emptyset$ and $G = \{a = b\}$. This topmost rewrite theory satisfies the condition of the previous theorem except G being topmost. Then, given the term $f(a, a)$, $f(b, a) =_G f(a, a)$ but now $f(b, a)$ cannot be rewritten with R.

For our bakery example, we can obtain a bisimilar equational abstraction of the folded transition system by restricting the abstraction only to the following equation, which intuitively collapses *extra* waiting processes that does not introduce any new behaviors:

```
eq (s s s L M) ; M ; PS0 [wait(s L M)] [wait(s s L M)]
 =   (s s L M) ; M ; PS0 [wait(s L M)] .
```

▶ **Lemma 18.** *The above equation satisfies the bisimilarity conditions (*) in Theorem 17.*

Proof. If we consider the rule `[wake]: N ; M ; [idle] PS => (s N) ; M ; [wait(N)] PS`, then $CSU_E(l = u)$ has the single E-unifier $\sigma = \{\mathtt{PS} \mapsto \mathtt{PS}_1\mathtt{[wait(s\,M\,L)]\,[wait(s\,s\,M\,L)]}, \mathtt{N} \mapsto \mathtt{s\,s\,s\,M\,L}, \mathtt{PS}_0 \mapsto \mathtt{PS}_1\mathtt{[idle]}\}$, where E denotes the equational axioms. Then, $\sigma v =_E \theta l$ and $\sigma r =_{E \cup G} \theta r$, for the substitution $\theta = \{\mathtt{N} \mapsto \mathtt{s\,s\,M\,L}, \mathtt{PS} \mapsto \mathtt{PS}_1\mathtt{[wait(s\,M\,L)]}\}$. For the other direction of the equation, $CSU_E(l = v) = \big\{\sigma' = \{\mathtt{PS} \mapsto \mathtt{PS}_2\mathtt{[wait(s\,M\,L)]}, \mathtt{PS}_0 \mapsto \mathtt{PS}_2\mathtt{[idle]}, \mathtt{N} \mapsto \mathtt{s\,s\,M\,L}\}\big\}$, and for the substitution $\theta' = \{\mathtt{N} \mapsto \mathtt{s\,s\,s\,M\,L}, \mathtt{PS} \mapsto \mathtt{PS}_2\mathtt{[wait(s\,M\,L)]\,[wait(s\,s\,M\,L)]}\}$, we have $\sigma' u =_E \theta' l$ and $\sigma' r =_{E \cup G} \theta' r$. The cases for the other rules are similar. ◀

Therefore, in order to model check the invariant $\Box$`ex?` from the initial pattern `N ; N ; IS` for an unbounded number of processes, we can then construct the *finite* abstract folded logical transition system from the given initial pattern displayed in Figure 5, where any counterexamples found are *not* spurious by Theorems 10 and 16.

4 Logical Bounded LTL Model Checking with Folding

We have shown that an infinite *logical* state space can be reduced to a finite state space using folding abstractions and equational abstractions. Although we can always achieve a finite logical state space using a *trivial* equational abstraction that collapses every state into a single state, the interesting case is obtaining *bisimilar* equational abstractions that produce a finite logical state space. However, we cannot ensure *a priori* whether such an abstract logical state space is finite or not, since it is in general undecidable for many infinite-state systems. Therefore, we introduce a logical bounded model checking (LBMC) method for verifying LTL properties, which provides an under-approximation of a logical state space.

In LBMC, we construct a *k-step folding abstraction* of $\mathcal{K}$ whose states are reachable in k-steps from a set of initial states $I \subseteq S$. Such a depth k is iteratively incremented until a certain bound or until reaching a fixed-point if it exists.

▶ **Definition 19.** Given $\mathcal{K} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $I \subseteq S$, the *k-step folding abstraction* of $\mathcal{K}$ from I is the Kripke structure

$$\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq,k}(I) = (Post_{\mathcal{K}\preccurlyeq}^{\leq k}(I),\ AP,\ \mathcal{L},\ \longrightarrow_{\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq,k}(I)}),$$

where $Post_{\mathcal{K}\preccurlyeq}^{\leq k}(I) = \bigcup_{0\leq i\leq k} Post_{\mathcal{K}\preccurlyeq}^{i}(I)$ and $\longrightarrow_{\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq,k}(I)} = \bigcup_{0\leq i\leq k} \longrightarrow_{\mathcal{K},i}^{\preccurlyeq}$.

For a (∞-step) folding abstraction $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$ we can easily see that if its state set $Post_{\mathcal{K}\preccurlyeq}^{*}(I)$ is finite, there exists $n \in \mathbb{N}$ such that $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq,j}(I) = \mathcal{R}each_{\mathcal{K}}^{\preccurlyeq}(I)$ for any $j \geq n$ by definition. Therefore, unlike typical bounded model checking methods (e.g., [3]), our folding-based method can easily detect if $\mathcal{R}each_{\mathcal{K}}^{\preccurlyeq,n}(I)$ is *complete* or not.

The LBMC of a logical Kripke structure $\mathcal{N}$ with a set of initial states I and a folding preorder $\preccurlyeq$ consists in model checking $\mathcal{N}(I)_i^{\preccurlyeq} = \mathcal{R}each_{\mathcal{N}}^{\preccurlyeq,i}(I)$ for each $i \in \mathbb{N}$, iteratively from 0 until one of the following termination conditions holds: (i) $\mathcal{N}(I)_i^{\preccurlyeq}$ is complete (a fixpoint is found), (ii) a counterexample is found in $\mathcal{N}(I)_i^{\preccurlyeq}$, or (iii) i is greater than a given maximum bound n. The LBMC algorithm for an LTL formula φ is briefly described as follows:

1. Apply a standard explicit-state LTL model checking algorithm to verify φ on $\mathcal{N}(I)_k^{\preccurlyeq}$. If a counterexample of φ is found in $\mathcal{N}(I)_k^{\preccurlyeq}$, stop and return the counterexample.
2. Suppose that there is *no* counterexample of φ in $\mathcal{N}(I)_k^{\preccurlyeq}$.
   a. If $k \geq n$, stop and report that $\mathcal{N}$ does not violate φ until the current bound k.
   b. Otherwise, generate $\mathcal{N}(I)_{k+1}^{\preccurlyeq}$ with the next bound $k+1$:
      i. If $\mathcal{N}(I)_{k+1}^{\preccurlyeq}$ is identical to $\mathcal{N}(I)_k^{\preccurlyeq}$, that is, $\mathcal{N}(I)_k^{\preccurlyeq}$ is complete, return *true*;
      ii. Otherwise, increment the depth-bound k by 1 and go to Step 1.

If the LBMC algorithm returns a counterexample, there are three possibilities according to the underlying folding preorder. If $\preccurlyeq$ is the E-renaming equivalence $\approx_E$ or φ is an invariant, it is an actual counterexample in $\mathcal{N}$. If $\preccurlyeq$ is the E-subsumption $\preccurlyeq_E$ and φ is a general LTL formula, it may be a spurious counterexample. Of course, if an equation abstraction has been applied, it is a real counterexample in $\mathcal{N}$ only for a bisimilar equational abstraction. Note that the above LBMC algorithm can easily be extended to guarantee the faithfulness for *safety* LTL properties as explained in Section 3.2.

5 The Maude LTL Logical Model Checker and Examples

This section illustrates the Maude LTL logical model checker (LMC) tool with two examples. This tool uses the existing narrowing framework in Full Maude to compute *narrowing* $\leadsto_{\sigma,R,E}$ [12]. However, for efficiency reasons, the core algorithms for the folding graph construction and the LTL model checking are implemented at the C++ level within the Maude system. For the LBMC algorithm, we apply an on-the-fly technique to reuse the previously generated states for the next step. The Maude LTL LMC tool and a number of other examples can be found in `http://formal.cs.illinois.edu/kbae/lmc`.

Our tool provides the following two commands for logical model checking an LTL formula φ from an initial state t with the maximum bound $n \in \mathbb{N}$:

`(lmc [`n`] `t` |= `φ` .)` and `(lfmc [`n`] `t` |= `φ` .)`

This bound n limits the depth of the k-step folding graph $\mathcal{R}each^{\preccurlyeq,k}_{\mathcal{N}^{AP}_{\mathcal{R}}}([t]_E)$ from an initial state $[t]_E \in \mathcal{N}^{AP}_{\mathcal{R}}$. Each command uses a different folding relation $\preccurlyeq$: the *renaming equivalence* $\approx_E$ for the `lmc` command, and the *subsumption* $\preccurlyeq_E$ for the `lfmc` command. If a bound n is not specified in the command, infinity is considered as the bound.

5.1 The Bakery Algorithm Revisited

The following command partially verifies that the mutual execution $\Box$`ex?` is satisfied from any initial state with the pattern `N ; N ; IS:ProcIdleSet` within the bound 10:

```
Maude> (lmc [10] N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex? .)
logical model check in BAKERY-SATISFACTION :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result:
  no counterexample found within bound 10
```

This model checking command does not terminate if the bound is not specified, since $\approx_E$ is not strong enough to collapse the reachable transition system to a finite one. The bound should be specified to ensure the termination even with $\preccurlyeq_E$, since, as already shown in Figure 4, for such a logical initial state the folding logical approximation is infinite:

```
Maude> (lfmc [50] N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex? .)
logical folding model check in BAKERY-SATISFACTION :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result:
  no counterexample found within bound 50
```

Instead, when the subsumption $\preccurlyeq_E$ is applied, with the bisimilar equational abstraction shown in Section 3.3, the mutual exclusion property $\Box$`ex?` can be verified from the initial pattern `N ; N ; IS:ProcIdleSet` as follows,[1] where, as shown in Figure 5, five logical states are generated in less than one second on an Intel Core i5 2.4 GHz with 4GB RAM:

```
Maude> (lfmc N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex? .)
logical folding model check in BAKERY-SATISFACTION-ABS :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result:
  true
```

[1] Note that the module `BAKERY-SATISFACTION-ABS` extends the previous module `BAKERY-SATISFACTION` with the abstraction equation in Section 3.3.

```
l0 : repeat
l1 :   flag[i] := 1
l2 :   while turn ≠ i do
         if flag[turn] = 0 then turn := i
l3 :   flag[i] := 2
l4 :   for j ≠ i do
         if flag[j] = 2 then goto l1
crit : /* critical region */
l5 :   flag[i] := 0
       forever
```

Figure 6 The Dijkstra's Mutual Exclusion Algorithm (for a process i) [27]

5.2 Dijkstra's Mutual Exclusion Algorithm

This section illustrates a topmost rewrite theory with another mutual exclusion algorithm for an arbitrary number of processes. Dijkstra's algorithm [27] considers n processes with $n \geq 2$, and two shared variables: (i) $flag[1 \ldots n]$ is an array of values $\{0, 1, 2\}$ for each process $1 \leq i \leq n$, and (ii) $turn$ is an integer between 1 and n. The behavior of this algorithm is summarized by the pseudo code in Figure 6.

We represent a state of this system as a multiset of triples $< \{f_1, p_1, t_1\} \cdots \{f_k, p_k, t_k\} >$, where each $\{f_i, p_i, t_i\}$ represents a process with f_i a value of $flag[i]$, p_i a program counter, and t_i a turn specifier that can be either `on` (i.e., $turn = i$) or `off` (i.e., $turn \neq i$). Only one process can be turned on at a time. The behavior of this system is then specified by the following topmost rewrite rules, where `PS` stands for the remaining set of precesses, and `WAITPS` stands for a multiset of processes whose flag is either 0 or 1:

```
rl [l1] : < {F,l0,T} PS >              => < {1,l1,T} PS > .
rl [l2] : < {F,l1,off} {0,S,on} PS >   => < {F,l1,on} {0,S,off} PS > .
rl [l2'] : < {F,l1,on} PS >            => < {F,l2,on} PS > .
rl [l3] : < {F,l2,T} PS >              => < {2,l3,T} PS > .
rl [l4] : < {F,l3,T} {2,S,T'} PS >     => < {1,l1,T} {2,S,T'} PS > .
rl [l4'] : < {F,l3,T} WAITPS >         => < {F,crit,T} WAITPS > .
rl [l5] : < {F,crit,T} PS >            => < {0,l5,T} PS > .
rl [l0] : < {F,l5,T} PS >              => < {F,l0,T} PS > .
```

Similar to the Bakery example, the mutual exclusion property of a single state can be specified by the atomic proposition `ex?`, defined by the following equations, where the logical variable `NCPS` stands for a set of processes whose program counter is *not* `crit`:

```
eq < NCPS > |= ex? = true .
eq < {F,crit,T} NCPS > |= ex? = true .
eq < {F,crit,T} {F',crit,T'} PS > |= ex? = false .
```

This system is infinite-state since the number of processes is unbounded. As a result, if the logical variable `IS` denotes a set of processes with flag 0 and program counter `l0`, the reachable logical state space from the pattern `< IS:InitProcSet >` is infinite even with the subsumption $\preccurlyeq_E$. However, we can obtain a *finite* bisimilar equational abstraction by adding the following topmost equations,[2] which satisfy the conditions (*) in Theorem 17:

```
eq < {F,l1,off} {F,l1,off} PS > = < {F,l1,off} PS > .
eq < {F,l5,off} {F,l5,off} PS > = < {F,l5,off} PS > .
```

[2] These equations G do *not* satisfy the finite variant property (see the conditions on [17]). However, all the *reachable* logical states from the given initial state `< IS >` have a finite set of most general G-variants, which is enough to have a finitary G-unification procedure for the *reachable* logical state space.

Then, the mutual exclusion □ex? can be verified from the pattern < IS:InitProcSet > that represents an arbitrary number of processes by the following command, where 42 logical spaces are generated in less than 6 second on the same machine:

```
Maude> (lfmc < IS:InitProcSet > |= [] ex? .)
logical folding model check in DIJKSTRA-MUTEX-SATISFACTION-ABS:
  < IS:InitProcSet > |= [] ex?
result:
  true
```

6 Conclusions

Using narrowing to model check LTL formulas on infinite-state systems is an intriguing symbolic method proposed in [16], whose practical effectiveness has required finding new methods to solve several open problems —such as Problems (1)–(3) in Section 1— and demonstrating its effectiveness in practice by supporting tools and examples. This paper has presented several new methods solving, or substantially improving, many of these difficulties, and the new Maude LTL logical model checker supporting these new techniques. We have also shown the effectiveness of the tool in verifying two nontrivial examples.

As usual much work remains ahead. Although the execution times shown in examples are quite reasonable, the tool's efficiency can be substantially improved by systematically exploiting the folding variant narrowing and unification features implemented at the C++ level in the upcoming new release of Maude. We plan to do this in the near future. Also, although the tool implementation is reasonably mature and has been tested on a collection of nontrivial examples, more experimentation is needed to increase its performance and illustrate its use on a wider set of applications. Another promising research direction is combining narrowing and SMT solving, using the *rewriting modulo SMT* ideas [33].

Acknowledgments. This work has been partially supported by NSF Grant CCF 09-05584, and for Santiago Escobar by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2010-21062-C02-02 and by Generalitat Valenciana PROMETEO2011/052.

References

1 P. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *CAV*, pages 452–466. Springer, 2002.
2 C. Baier and J. P. Katoen. *Principles of Model Checking*. The MIT Press, 2007.
3 A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.
4 A. Bouajjani and J. Esparza. Rewriting models of boolean programs. *Term Rewriting and Applications*, pages 136–150, 2006.
5 A. Bouajjani and T. Touili. Widening techniques for regular tree model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):145–165, 2012.
6 T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *CAV*, pages 400–411. Springer, 1997.
7 O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. *Handbook of Process algebra*, pages 545–623, 2001.
8 E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2001.
9 M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

10 H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. *Term Rewriting and Applications*, pages 294–307, 2005.

11 G. Delzanno and A. Podelski. Constraint-based deductive model checking. *STTT*, 3, 2001.

12 F. Durán, S. Eker, S. Escobar, J. Meseguer, and C. Talcott. Variants, unification, narrowing, and symbolic reachability in maude 2.6. In *RTA*, volume 10, pages 31–40, 2011.

13 F. Durán and J. Meseguer. A maude coherence checker tool for conditional order-sorted rewrite theories. In *Rewriting Logic and Its Applications*. Springer, 2010.

14 E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Logic in Computer Science*, pages 70–80. IEEE, 1998.

15 S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 1–50. Springer, 2009.

16 S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *RTA*, pages 153–168, 2007.

17 S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *J. Algebraic and Logic Programming*, 81:898–928, 2012.

18 A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. *Algebraic Methodology and Software Technology*, pages 142–157, 2006.

19 A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

20 F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems: An experimental evaluation. In *Logic-Based Program Synthesis and Transformation*, volume 6564 of *LNCS*, pages 164–183. Springer, 2010.

21 T. Genet and V. Rusu. Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5):574–597, 2010.

22 T. Genet and V. Tong. Reachability analysis of term rewriting systems with timbuk. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2001.

23 J. M. Hullot. Canonical forms and unification. In *CADE*, LNCS vol. 87. Springer, 1980.

24 J. P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proc. ICALP*, volume 154 of *LNCS*. Springer, 1983.

25 Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *STTT*, 2(4):328–342, 2000.

26 O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

27 Nancy Ann Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

28 J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

29 J. Meseguer. Membership algebra as a logical framework for equational specification. In *WADT*, volume 1376 of *LNCS*, pages 18–61. Springer, 1997.

30 J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.

31 J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.

32 H. Ohsaki, H. Seki, and T. Takai. Recognizing boolean closed a-tree languages with membership conditional rewriting mechanism. In *RTA*, pages 483–498. Springer, 2003.

33 C. Rocha. *Symbolic Reachability Analysis for Rewrite Theories*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

34 A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. *TACAS*, pages 45–60, 2005.

Compression of Rewriting Systems for Termination Analysis * †

Alexander Bau[1], Markus Lohrey[2], Eric Nöth[2], and Johannes Waldmann[1]

1 **HTWK Leipzig, Germany**
{abau,waldmann}@imn.htwk-leipzig.de
2 **Universität Leipzig, Germany**
{lohrey,noeth}@informatik.uni-leipzig.de

Abstract

We adapt the TreeRePair tree compression algorithm and use it as an intermediate step in proving termination of term rewriting systems. We introduce a cost function that approximates the size of constraint systems that specify compatibility of matrix interpretations. We show how to integrate the compression algorithm with the Dependency Pairs transformation. Experiments show that compression reduces running times of constraint solvers, and thus improves the power of automated termination provers.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases termination of rewriting, matrix interpretations, constraint solving, tree compression

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.97

1 Introduction

For proving termination of rewrite systems automatically, a standard approach is to use monotone interpretations, e.g., by polynomials [11] or matrices [6] for the function symbols. The conditions of monotonicity and of compatibility with a given rewrite system result in a constraint system for the coefficients of the interpretation. Termination provers then use a constraint solver to obtain an actual interpretation. One way of solving constraint systems is bit-blasting [7, 13, 4]: The unknown numerical coefficients are represented by sequences of boolean unknowns, the constraints are transformed to a formula in propositional logic, and a state-of-the-art SAT solver [5] is applied to find a satisfying assignment, from which the interpretation can be reconstructed.

In the present paper, we describe and investigate a method that allows to obtain small constraint systems for the termination problem of a given rewrite system. From a given rewrite system we compute a *straight line program* that produces all left-hand and right-hand sides of the rewrite system. The elementary operation of this straight line program is *substitution* of terms. Such straight line programs were used for tree compression in [12]. The computed straight line program can be directly seen as a straight line program that computes the coefficients of the linear interpretations for all left-hand and right-hand sides from the unknown coefficients of the function symbols. The elementary operations

* The first author is supported by an European Social Fund grant.
† The second and third author are supported by the DFG research project ALKODA.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 97–112

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of this new straight line program are matrix-matrix and matrix-vector multiplications. We define the cost of the straight line program as the number of matrix-matrix multiplications, which is justified by the fact that matrix-vector multiplications are cheap in comparision to matrix-matrix multiplications.

Hence, our goal is to compute from a given rewrite system a straight line program with small cost. We do this by an adaptation of the TreeRePair algorithm [12] for tree compression. In each iteration, TreeRePair finds a most frequently occurring *digram* (a term pattern consisting of two function symbols) in the input tree and replaces this digram by a fresh symbol. For the purpose of proving termination with matrix interpretations, TreeRePair has been used in [6] (where a naive implementation of TreeRePair has been described independently from [12]). Our new adaptation of TreeRePair chooses in each iteration a digram that reduces the cost (number of matrix multiplications) maximally.

Our cost function (number of matrix multiplications) directly relates to the size of the constraint system (size of the CNF formula). This is a simple monotonic relation. It depends on the number of matrix constraints, the dimensions of the matrices, and the encoding of the operations on the matrix elements. In the present paper, we omit the discussion of elementary operations, and refer to [3] instead. The size of the formula relates, in turn, to the running time of the SAT solver. In general this relation is not monotonic, and it seems impossible to predict the relation exactly, since it depends on the particulars of simplification and resolution strategies embedded in the SAT solver, which is outside the scope of this paper. We therefore just assume that smaller formulas give, in general, shorter running times for solvers, and we test this hypothesis by experiments.

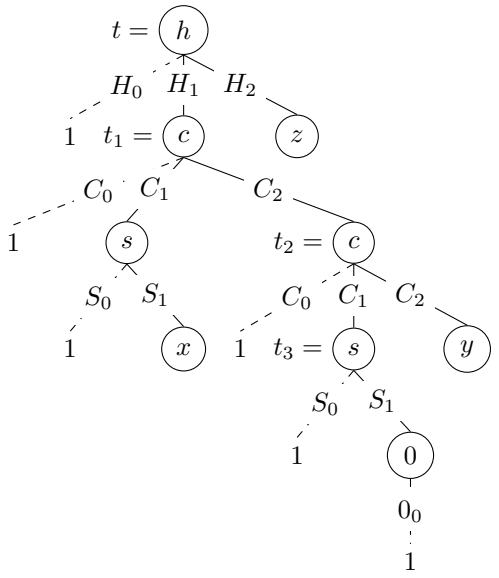
The new contributions of the present paper are:

- We define a cost function for terms (digrams) that reflects the size of constraint systems more accurately, by taking occurrences of variables in subterms into account.
- We present the algorithm MCTreeRePair that finds optimal digrams and discuss its performance.
- We combine our approach with the dependency pairs transformation [1].
- We present an open-source implementation of our algorithm, and we give an experimental evaluation, when applied as part of a realistic termination prover.

2 Terms

We use standard notations for terms and term rewriting systems. Let Σ be a finite *ranked alphabet of symbols* (also called a *signature*), where the *rank* or *arity* of $f \in \Sigma$ is denoted with $\mathrm{rk}(f) \in \mathbb{N}$. Let $\Sigma_l = \{f \in \Sigma \mid \mathrm{rk}(f) = l\}$ for $l \in \mathbb{N}$ and let k be maximal such that $\Sigma_k \neq \emptyset$. A *term* (or *tree*) over Σ is a pair $t = (D, \lambda)$ where D is a finite prefix closed and non-empty subset of $\{1, \ldots, k\}^*$ and λ is a function from D to Σ such that for all $p \in D$ and $1 \leq d \leq k$: $pd \in D$ if and only if $1 \leq d \leq \mathrm{rk}(\lambda(p))$. Elements of D are also called *positions* or *nodes* of t. Define $|t| = |D|$ (the size of the term t). For $p \in D$ let $t|_p = (\{q \in \{1, \ldots, k\}^* \mid pq \in D\}, \lambda')$, where $\lambda'(q) = \lambda(pq)$ for $pq \in D$, be the *subterm* of t rooted at position p. We write $s \trianglelefteq t$ (resp. $s \triangleleft t$) if s is a subterm (resp., a proper subterm) of t. With $\mathrm{Term}(\Sigma)$ we denote the set of all terms over Σ. We use the standard term notation, i.e., if for $t = (D, \lambda)$ we have $\lambda(\varepsilon) = f$, $\mathrm{rk}(f) = n$, and $t_i = t|_i$ for $1 \leq i \leq n$, then $t = f(t_1, \ldots, t_n)$.

Let V be a finite set of *variables* with $\Sigma \cap V = \emptyset$. We define $\mathrm{Term}(\Sigma, V) = \mathrm{Term}(\Sigma \cup V)$, where every variable $x \in V$ gets rank 0, i.e., is treated as a constant. For a term $t \in \mathrm{Term}(\Sigma, V)$ let $\mathrm{Var}(t)$ be the set of variables that occur at least once in t. A *term rewriting system* (TRS) over the signature Σ is a finite set $R \subseteq \mathrm{Term}(\Sigma, V) \times \mathrm{Term}(\Sigma, V)$ such that



coefficients of variables:

$$\begin{aligned}
\mathrm{coeff}_x(t_1) &= C_1 \cdot S_1 \\
\mathrm{coeff}_y(t_1) &= C_2 \cdot C_2 \\
\mathrm{coeff}_x(t) &= H_1 \cdot \mathrm{coeff}_x(t_1) \\
\mathrm{coeff}_y(t) &= H_1 \cdot \mathrm{coeff}_y(t_1)
\end{aligned}$$

absolute parts:

$$\begin{aligned}
\mathrm{coeff}_1(t_3) &= S_1 \cdot 0_0 \\
\mathrm{coeff}_1(t_2) &= C_0 + C_1 \cdot \mathrm{coeff}_1(t_3) \\
\mathrm{coeff}_1(t_1) &= C_0 + C_1 \cdot S_0 + C_2 \cdot \mathrm{coeff}_1(t_2) \\
\mathrm{coeff}_1(t) &= H_0 + H_1 \cdot \mathrm{coeff}_1(t_1)
\end{aligned}$$

Figure 1 Bottom-up computation of the coefficient matrices of $[h(c(s(x), c(s(0), y)), z)]$.

for every rule $(\ell \to r) \in R$ we have $\ell \notin V$ and $\mathrm{Var}(r) \subseteq \mathrm{Var}(\ell)$. The *one-step rewriting relation* $\to_R$ is defined as usual.

3 Matrix interpretation of terms and the cost of terms

As explained in the introduction we use matrix interpretations for proving termination of term rewriting systems. To define such interpretations, let us fix a semiring S (the ring of matrix coefficients) and a dimension $n \geq 1$ for the further consideration. We want to interpret a term t with m different variables as an m-ary function from S^n to S^n. For a set of variables $U \subseteq V$ we denote with $(S^n)^U$ the set of all mappings from U to S^n, or alternatively, the set of U-indexed tuples over S^n. Moreover, for every symbol $f \in \Sigma_m$ we fix matrices $F_1, \ldots, F_m \in S^{n \times n}$ and a vector $F_0 \in S^n$. This allows us to define the linear function $[f] : (S^n)^m \to S^n$ by

$$[f](x_1, \ldots, x_m) = F_0 + F_1 x_1 + \cdots + F_m x_m, \tag{1}$$

where $x_1, \ldots, x_m \in S^n$. Now let $t \in \mathrm{Term}(\Sigma, V)$ be a term with $U = \mathrm{Var}(t)$. The interpretation $[t] : (S^n)^U \to S^n$ is computed by composing the interpretations for the ranked symbols in the natural way: Let $t = f(t_1, \ldots, t_k)$. Then $[t](\overline{x}) = [f]([t_1](\overline{x}_1), \ldots, [t_k](\overline{x}_k))$, where $\overline{x} \in (S^n)^U$ and $\overline{x}_i$ is the restriction of $\overline{x}$ to $\mathrm{Var}(t_i) \subseteq U$.

We next define a cost measure for terms that reflects the number of matrix multiplications that have to be done when the coefficients of the linear function that is represented by a term, are computed in a bottom-up manner. Recall that a term $t \in \mathrm{Term}(\Sigma, V)$ with $U = \mathrm{Var}(t)$ represents the linear function $[t] : (S^n)^U \to S^n$, which can be written as an expression $T_0 + T_1 x_1 + \cdots + T_k x_k$, where $\mathrm{Var}(t) = \{x_1, \ldots, x_k\}$, $T_0 \in S^n$ and $T_1, \ldots, T_k \in S^{n \times n}$. We identify $[t]$ with this expression. Moreover, let $\mathrm{coeff}_{x_i}(t) = T_i$. Figure 1 shows the bottom-up calculation of the coefficients of an example term (from Example 2 below). The constant term (F_0 in (1)) is denoted by $\mathrm{coeff}_1(\cdot)$. In total, the computation of $[t]$ needs 4 multiplications of an $(n \times n)$-matrix by an $(n \times n)$-matrix, and 5 multiplications of an $(n \times n)$-matrix

by an $(n \times 1)$-matrix (a vector). In the following, we will only count matrix-by-matrix multiplications, and ignore matrix-by-vector multiplications, as well as additions. This is justified by higher asymptotic cost of multiplications (n^3 versus n^2). The matrix dimension in our applications is small (at most 5), see Section 7. Hence, matrix multiplication algorithms with an asymptotic running time better than $O(n^3)$ (e.g., Strassen's algorithm) are not useful in our context.

Let $t = f(t_1, \ldots, t_k)$, and assume that all coefficient matrices $\mathrm{coeff}_x(t_i)$ are already known. Then we can compute the coefficient matrix $\mathrm{coeff}_x(t)$ as

$$\mathrm{coeff}_x(t) = \sum_{i=1}^{k} F_i \cdot \mathrm{coeff}_x(t_i),$$

where F_i is from (1), and we set $\mathrm{coeff}_x(t_i) = 0$ if $x \notin \mathrm{Var}(t_i)$. Hence, we have one matrix multiplication for each i with $x \in \mathrm{Var}(t_i)$. But note that the multiplication is trivial if $x = t_i$ (multiplication by the identity matrix, which costs nothing). This motivates the following definition:

▶ **Definition 1.** The *(matrix multiplication) cost* of a term $t = (D, \lambda) \in \mathrm{Term}(\Sigma, V)$ is

$$\mathrm{cost}(t) = \sum_{p \in D \setminus \{\varepsilon\}, \lambda(p) \notin V} |\mathrm{Var}(t|_p)|. \tag{2}$$

The cost of a tuple $(t_1, \ldots, t_m)$ of terms is $\sum_{i=1}^{m} \mathrm{cost}(t_i)$.

Note that this definition models a bottom-up evaluation where we do not use any caching, memoization, etc. The following example has been taken from [6].

▶ **Example 2.** Let $\mathrm{rk}(h) = \mathrm{rk}(c) = 2$, $\mathrm{rk}(s) = 1$, and $\mathrm{rk}(0) = 0$. Then we have

$$\mathrm{cost}(h(x, c(y, z))) = 2 \qquad \mathrm{cost}(h(c(s(x), c(s(0), y)), z)) = 4$$
$$\mathrm{cost}(h(c(s(y), x), z)) = 3 \qquad \mathrm{cost}(h(y, c(s(0), c(x, z)))) = 4$$

Figure 1 shows a detailed computation of the coefficients of the interpretation of the term $h(c(s(x), c(s(0), y)), z)$.

4 Term compression with TreeRePair

In this section we describe an adaptation of the RePair compression algorithm for strings, which is called TreeRePair in [12], see also [6]. The idea is to replace frequently occurring tree patterns of size 2 (so called *digrams*) by new symbols (called non-terminals in [12]) and to store in a table the patterns corresponding to the new symbols. Most of the notation from this section will be needed in the next section where we outline an adaptation of TreeRePair with the goal of reducing the matrix multiplication cost of a list of terms.

Let us fix a ranked alphabet Σ.

▶ **Definition 3.** A *digram* over Σ is a triple $d = [f, i, g]$ where $f, g \in \Sigma$ and $1 \le i \le \mathrm{rk}(f))$. The *rank* (or arity) of this digram is $\mathrm{rk}(d) = \mathrm{rk}(f) - 1 + \mathrm{rk}(g)$.

We consider a digram d as a new symbol of rank $\mathrm{rk}(d)$.

▶ **Definition 4.** To the digram $d = [f, i, g]$ with $\mathrm{rk}(d) = n$ and $\mathrm{rk}(g) = l$ we associate the rewrite rule

$$\mathsf{rule}(d) = (d(x_1, \ldots x_n) \to f(x_1, \ldots x_{i-1}, g(x_i, \ldots, x_{i+l-1}), x_{i+l}, \ldots x_n)).$$

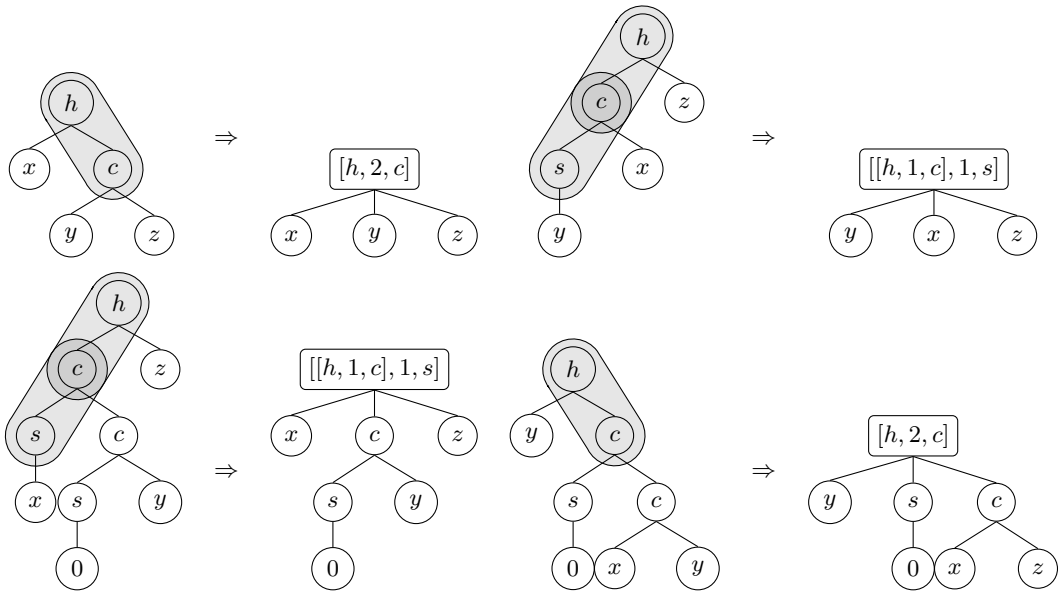


Figure 2 The replaced digrams from Example 7.

With $\mathsf{rule}(d)^{-1}$ we denote the reverse rule

$$f(x_1, \dots x_{i-1}, g(x_i, \dots, x_{i+l-1}), x_{i+l}, \dots x_n)) \to d(x_1, \dots x_n).$$

The right-hand side of the rule $\mathsf{rule}(d)$ can be seen as the tree pattern represented by the digram d.

▶ **Definition 5.** A *compressed term list* is a list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$, where

- for each $1 \le i \le n$, d_i is a digram over the signature $\Sigma \cup \{d_1, \dots, d_{i-1}\}$, and
- for each $1 \le i \le m$, $t_i \in \mathrm{Term}(\Sigma \cup \{d_1, \dots, d_n\}, V)$.

The idea is that a compressed term list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ represents the term list $(s_1, \dots, s_m)$ that is obtained by replacing nodes labeled with digram symbols by the corresponding tree patterns (of size 2). This motivates the following definition:

▶ **Definition 6.** The *expansion* of a compressed term list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ is the list $(s_1, \dots, s_m)$ where s_i is the unique normal form of t_i w.r.t. to the (confluent and terminating) term rewriting system $\{\mathsf{rule}(d_1), \dots, \mathsf{rule}(d_n)\}$.

▶ **Example 7.** Let $\mathrm{rk}(h) = \mathrm{rk}(c) = 2$, $\mathrm{rk}(s) = 1$, $\mathrm{rk}(0) = 0$, and consider the following compressed term list:

$$([h,2,c](x,y,z),\ [[h,1,c],1,s](y,x,z),\ [[h,1,c],1,s](x,c(s(0),y),z),\ [h,2,c](y,s(0),c(x,z)) \mid [h,1,c],\ [h,2,c],\ [[h,1,c],1,s])$$

The expansion of this list is the following term list consisting of the terms from Example 2:

$$(h(x,c(y,z)),\ h(c(s(y),x),z),\ h(c(s(x),c(s(0),y)),z),\ h(y,c(s(0),c(x,z)))). \tag{3}$$

Figure 2 shows the replaced digrams from the above list.

In our applications, a term list will be a list of all left-hand and right-hand sides of a TRS. If term (list) compression is the main objective, then the goal would be to compute a small compressed term list whose expansion is the input term list. For this let us define the size of a compressed term list $(t_1, \ldots, t_m \mid d_1, \ldots, d_n)$ as $\sum_{i=1}^{m} |t_i| + n$. This definition is justified by the fact that a digram can be stored by two symbols (either symbols from the initial signature or references to previously defined digrams) and an integer, which needs constant space in a uniform cost model. In Example 7 the compressed term list has size 25, whereas the expanded term list has size 28.

From the results of [2] it follows immediately that it is NP-complete to check whether for a given term list $\overline{t} = (t_1, \ldots, t_m)$ and a given number k there exists a compressed term list of size at most k whose expansion is $\overline{t}$. This holds even for the special case that all symbols in $\overline{t}$ are unary and $m = 1$ (in this case $\overline{t} = t_1$ is basically a string and a compressed term list is a context-free grammar in Chomsky normal form that only produces t_1).

In [12], the algorithm TreeRePair for producing a small compressed term list was presented. Let us briefly explain the idea, for which we need a few definitions. Fix a digram $d = [f, i, g]$ and a term list $\overline{t} = (t_1, \ldots, t_m)$, where $t_j = (D_j, \lambda_j)$. An *occurrence* of d in $\overline{t}$ is a pair (j, p) where $1 \leq j \leq m$, $p \in D_j$, $\lambda_j(p) = f$, and $\lambda_j(pi) = g$. A set Occ of occurrences of d in $\overline{t}$ is *non-overlapping* if for every $(j, p) \in \mathsf{Occ}$ we have $(j, pi) \notin \mathsf{Occ}$. Clearly, if $f \neq g$, then every set of occurrences is non-overlapping. If Occ is non-overlapping then we can apply in each term t_j simultaneously at all positions p with $(j, p) \in \mathsf{Occ}$ the rewrite rule $\mathsf{rule}(d)^{-1}$. Let t'_j be the resulting term. We write $(t_1, \ldots, t_m) \to_{\mathsf{Occ}} (t'_1, \ldots, t'_m)$.

Let $\mathsf{max}(d, \overline{t})$ be the maximal size among all non-overlapping sets of occurrences of d in $\overline{t}$. We can easily determine a non-overlapping set $\mathsf{maxOcc}(d, \overline{t})$ of occurrences of d in $\overline{t}$ such that $|\mathsf{maxOcc}(d, \overline{t})| = \mathsf{max}(d, \overline{t})$: If $f \neq g$ then we can set $\mathsf{maxOcc}(d, \overline{t})$ to the set of all occurrences of d in $\overline{t}$. On the other hand, if $f = g$, then we obtain $\mathsf{maxOcc}(d, \overline{t})$ as follows. For every maximal chain of positions $p, pi, pii, \ldots, pi^k \in D_j$ in a term $t_j = (D_j, \lambda_j)$ from our list such that $\lambda_j(pi^\ell) = f$ for all $0 \leq \ell \leq k$ we do the following: If k is odd, then we add the pairs $(j, p), (j, pi^2), \ldots, (j, pi^{k-1})$ to $\mathsf{maxOcc}(d, \overline{t})$. If k is even, then we add the pairs $(j, p), (j, pi^2), \ldots, (j, pi^{k-2})$ to $\mathsf{maxOcc}(d, \overline{t})$. In the case that k is even we could have also put the pairs $(j, pi), (j, pi^3), \ldots, (j, pi^{k-1})$ into $\mathsf{maxOcc}(d, \overline{t})$; this choice would also lead to an occurrence set of size $\mathsf{max}(d, \overline{t})$. We prefer $(j, p), (j, pi^2), \ldots, (j, pi^{k-2})$ since this is the better choice for our adaptation MCTreeRePair of TreeRePair described in Section 5. A high-level description of the TreeRePair algorithm looks as follows:

input: a term list $\overline{t} = (t_1, \ldots, t_m)$
$\overline{d} := \varepsilon$ (a list of digrams)
while there exists a digram d with $\mathsf{max}(d, \overline{t}) > 1$ **do**
  let d be a digram with $\mathsf{max}(d, \overline{t}) \geq \mathsf{max}(d', \overline{t})$ for all digrams d'
  let $\overline{u}$ such that $\overline{t} \to_{\mathsf{maxOcc}(d, \overline{t})} \overline{u}$
  $\overline{t} := \overline{u}$; $\overline{d} := (\overline{d}, d)$
endwhile
output: $(\overline{t} \mid \overline{d})$

In [12] the user of TreeRePair may specify a parameter r with the following meaning: Only digrams d with $\mathrm{rk}(d) \leq r$ will be considered in each iteration of the algorithm. This has two advantages:

- For the test data in [12] (large XML tree structures) it leads to a better compression ratio, see [12] for an explanation of this possibly surprising fact. The optimal value turned out to be $r = 4$.

- Bounding the maximal rank of digrams improves the running time of TreeRePair drastically.

Let us briefly explain the second point: A naive implementation of the above algorithm, where we count digram frequencies in each iteration of the while-loop from scratch needs quadratic time. The implementation from [12] avoids this; thereby some occurrences of self-overlapping digrams of the form $[f, i, f]$ get lost (as explained at the end of this section). But the resulting negative effect on the compression ratio turned out to be negligible. We recall some implementation details from [12], since our implementation of MCTreeRePair uses the same principles.

The input terms from $\bar{t}$ are represented as pointer structures, where every node stores a pointer to its parent node and a list of pointers to its children. An occurrence (j, p) of d in $\bar{t}$ is simply represented by a pointer to node p of t_j. Initially, for every digram d all occurrences from the set $\mathsf{maxOcc}(d, \bar{t})$ are inserted into a doubly linked list (one for each digram) and the size of $\mathsf{maxOcc}(d, \bar{t})$ (which is $\mathsf{max}(d, \bar{t})$) is counted. This can be done in a single pass over the input term list $\bar{t}$. If the total number of nodes in the input term list $\bar{t}$ is n then clearly at most n digram replacements can be done in total. Each time an occurrence (j, p) of the digram $d = [f, i, g]$ is replaced, the following steps are done:

- Delete the node pi of t_j, set the parent pointer for every children pik $(1 \leq k \leq \mathrm{rk}(g))$ of pi to p, insert the list of child pointers of node pi into the list of child pointers of node p, and change the label of p to d.
- Decrement the count-value for digrams d' that overlap the replaced occurrence of d at position p, and remove the overlapping occurrences of d' from the current list of d'-occurrences.
- Increment the count-value for those digrams d' that are introduced by the replacement step (digrams of the form $[h, l, d]$ or $[d, l, h]$), and insert new occurrences of d' into the list of d'-occurrences.

Assume that $\mathrm{rk}(f) = m$ and $\mathrm{rk}(g) = n$. Then at most $m + n$ digram occurrences overlap the replaced occurrence of d at position p. The rank of d is $m + n - 1$. Hence, if we only replace digrams of rank at most r, then at most $r + 1$ digram occurrences overlap the replaced occurrence of d at position p. Hence, per digram replacement only a constant number of updates is necessary.

A problem arises with self-overlapping digrams of the form $[f, i, f]$. Consider for instance the term $f(a, f(b, f(c, d)))$. The occurrence of $[f, 2, f]$ at the root position ε would belong to the set $\mathsf{maxOcc}([f, 2, f], \bar{t})$, whereas the second occurrence at position 2 would not. Now assume that we replace the digram $[f, 1, a]$ (by adding further terms to the term list, $[f, 1, a]$ might become the most frequent digram). We obtain the term $A_1(f(b, f(c, d)))$. If we would compute the set $\mathsf{maxOcc}([f, 2, f], \bar{t})$ from scratch, the occurrence of $[f, 2, f]$ at position 1 in the term $A_1(f(b, f(c, d)))$ would be inserted into the set $\mathsf{maxOcc}([f, 2, f], \bar{t})$. But in the implementation from [12] described above this occurrence is lost.

5 Minimizing the matrix multiplication cost using TreeRePair

In this section we present our adaptation of TreeRePair with the goal of reducing the matrix multiplication cost of a given term list. We call our algorithm MCTreeRePair ("MC" for "matrix cost"). The point is that a compressed term list may have smaller cost than its expansion. First, we have to define the cost of a digram:

▶ **Definition 8.** The *cost* of digram $d = [f, i, g]$ is $\mathrm{cost}(d) = \mathrm{rk}(g)$.

Note that $\mathrm{cost}(d) = \mathrm{rk}(g)$ is exactly the number of matrix multiplications needed to compute the coefficients for the linear function that is represented by d.

▶ **Definition 9.** The *(matrix multiplication) cost* of a compressed term list $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ is

$$\mathrm{cost}(t_1, \dots, t_m \mid d_1, \dots, d_n) = \sum_{i=1}^{m} \mathrm{cost}(t_i) + \sum_{i=1}^{n} \mathrm{cost}(d_i). \tag{4}$$

If $(s_1, \dots, s_m)$ is the expansion of $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ then we can compute the coefficients for the linear functions $[s_1], \dots, [s_m]$ with $\mathrm{cost}(t_1, \dots, t_m \mid d_1, \dots, d_n)$ many matrix multiplications.

▶ **Example 10.** Let us compute the cost of the compressed term list from Example 7. We have:

$$\begin{aligned} \mathrm{cost}([h,2,c](x,y,z)) &= 0 & \mathrm{cost}([h,1,c]) &= 2 \\ \mathrm{cost}([[h,1,c],1,s](y,x,z)) &= 0 & \mathrm{cost}([h,2,c]) &= 2 \\ \mathrm{cost}([[h,1,c],1,s](x,c(s(0),y),z)) &= 1 & \mathrm{cost}([[h,1,c],1,s]) &= 1 \\ \mathrm{cost}([h,2,c](y,s(0),c(x,z))) &= 2 \end{aligned}$$

Hence, the total cost is 8. In contrast, the cost of the expanded term list in (3) is 13.

▶ **Definition 11.** The *savings* of a non-overlapping set of occurrences Occ of a digram $d = [f,i,g]$ in a term list $\bar{t} = (t_1, \dots, t_m)$, briefly $\mathsf{save}(\mathsf{Occ}, \bar{t})$, is defined as follows, where $t_j = (D_j, \lambda_j)$:

$$\mathsf{save}(\mathsf{Occ}, \bar{t}) = -\mathrm{cost}(d) + \sum_{(j,p) \in \mathsf{Occ}} |\mathrm{Var}(t_j|_{pi})|. \tag{5}$$

In other words: We add to the negative cost of d for each $(j,p) \in \mathsf{Occ}$ the number of different variables below the i-th child of node p (the lower digram node).

By the following lemma, $\mathsf{save}(\mathsf{Occ}, \bar{t})$ is exactly the cost-reduction obtained by replacing all digram occurrences from Occ.

▶ **Lemma 12.** *Let $(t_1, \dots, t_m \mid d_1, \dots, d_n)$ be a compressed term list, let $d = [f,i,g]$ be a digram, and let Occ be a non-overlapping set of occurrences of d in $(t_1, \dots, t_m)$. Let $(t_1, \dots, t_m) \to_{\mathsf{Occ}} (t'_1, \dots, t'_m)$. Then we have*

$$\mathrm{cost}(t'_1, \dots, t'_m \mid d_1, \dots, d_n, d) = \mathrm{cost}(t_1, \dots, t_m \mid d_1, \dots, d_n) - \mathsf{save}(\mathsf{Occ}, [t_1, \dots, t_m]). \tag{6}$$

Proof. Let $\mathsf{Occ}_j = \{p \mid (j,p) \in \mathsf{Occ}\}$. Using (4) and (5), it follows that (6) is equivalent to

$$\sum_{j=1}^{m} \mathrm{cost}(t_j) = \sum_{j=1}^{m} \mathrm{cost}(t'_j) + \sum_{(j,p) \in \mathsf{Occ}} |\mathrm{Var}(t_j|_{pi})|.$$

This follows from

$$\mathrm{cost}(t_j) = \mathrm{cost}(t'_j) + \sum_{p \in \mathsf{Occ}_j} |\mathrm{Var}(t_j|_{pi})|$$

for all $1 \le j \le m$. But this is a consequence of (2). Applying rule $\mathsf{rule}(d)^{-1}$ at all positions $p \in \mathsf{Occ}_j$ in t_j means that we remove all nodes pi with $p \in \mathsf{Occ}_j$ from t_j. Moreover, for all other nodes of t_j the number of different variables below the node does not change. Also note that for all $p \in \mathsf{Occ}_j$, node pi of t_j is not labeled with a variable. ◀

By Lemma 12, in order to reduce the cost of a (compressed) term list maximally, we have to find a non-overlapping set of occurrences (of some digram d) with maximal savings.

▶ **Definition 13.** For the digram $d = [f, i, g]$ and the term list $\overline{t} = (t_1, \dots, t_m)$ let $\mathsf{maxsave}(d, \overline{t})$ be the maximum of $\mathsf{save}(\mathsf{Occ}, \overline{t})$, where we maximize over all non-overlapping sets of occurrences Occ of d in $\overline{t}$.

Recall the definition of the non-overlapping occurrence set $\mathsf{maxOcc}(d, \overline{t})$ from Section 4.

▶ **Lemma 14.** *We have* $\mathsf{save}(\mathsf{maxOcc}(d, \overline{t}), \overline{t}) = \mathsf{maxsave}(d, \overline{t})$.

Proof. Let $d = [f, i, g]$. The case $f \neq g$ is clear, since then $\mathsf{maxOcc}(d, \overline{t})$ is the set of all occurrences of d in $\overline{t}$. Now assume that $f = g$. Recall that we obtain $\mathsf{maxOcc}(d, \overline{t})$ by considering all maximal chains of positions $p, pi, pii, \dots, pi^k \in D_j$ in a term $t_j = (D_j, \lambda_j)$ from our list such that $\lambda_j(pi^\ell) = f$ for all $0 \leq \ell \leq k$. If k is odd, then we put the occurrences $(j, p), (j, pi^2), \dots, (j, pi^{k-1})$ into $\mathsf{maxOcc}(d, \overline{t})$. If k is even, then we put the occurrences $(j, p), (j, pi^2), \dots, (j, pi^{k-2})$ into $\mathsf{maxOcc}(d, \overline{t})$. Note that in case k is even, the set of occurrences $\{(j, pi), (j, pi^3), \dots, (j, pi^{k-1})\}$ has the same size as the chosen set of occurrences $\{(j, p), (j, pi^2), \dots, (j, pi^{k-2})\}$. But the latter gives a larger (or the same) savings according to (5), since $\mathrm{Var}(t_j|_{pi^{\ell+1}}) \subseteq \mathrm{Var}(t_j|_{pi^\ell})$ and thus $|\mathrm{Var}(t_j|_{pi^{\ell+1}})| \leq |\mathrm{Var}(t_j|_{pi^\ell})|$ for all $0 \leq \ell \leq k-1$. ◀

On a high level, MCTreeRePair works as follows:

input: a term list $\overline{t} = (t_1, \dots, t_m)$
$\overline{d} := \varepsilon$ (a list of digrams)
while there exists a digram d with $\mathsf{maxsave}(d, \overline{t}) > 0$ **do**
  let d be a digram with $\mathsf{maxsave}(d, \overline{t}) \geq \mathsf{maxsave}(d', \overline{t})$ for all digrams d'
  let $\overline{u}$ such that $\overline{t} \to_{\mathsf{maxOcc}(d, \overline{t})} \overline{u}$
  $\overline{t} := \overline{u}$; $\overline{d} := (\overline{d}, d)$
endwhile
output: $(\overline{t} \mid \overline{d})$

Here is a complete example run of MCTreeRePair.

▶ **Example 15.** Let $\mathrm{rk}(h) = \mathrm{rk}(c) = 2$, $\mathrm{rk}(s) = 1$ and $\mathrm{rk}(0) = 0$. Consider the following term rewriting system (which consists of the terms from Example 2):

$$h(x, c(y, z)) \to h(c(s(y), x), z), \qquad h(c(s(x), c(s(0), y)), z) \to h(y, c(s(0), c(x, z)))$$

The matrix multiplication cost is 13, see Example 2. The $\mathsf{maxsave}$-values of the digrams in this system are (we omit the second parameter in $\mathsf{maxsave}$ for the term list):

$$\mathsf{maxsave}(h, 1, c) = 2, \quad \mathsf{maxsave}(c, 1, s) = 1, \quad \mathsf{maxsave}(s, 1, 0) = 0$$
$$\mathsf{maxsave}(h, 2, c) = 2, \quad \mathsf{maxsave}(c, 2, c) = 1,$$

Let us decide to replace the digram $d := (h, 1, c)$ (we could also choose $(h, 2, c)$). We obtain the following system:

$$h(x, c(y, z)) \to d(s(y), x, z), \qquad d(s(x), c(s(0), y), z) \to h(y, c(s(0), c(x, z)))$$

The new $\mathsf{maxsave}$-values are:

$$\mathsf{maxsave}(h, 2, c) = 2, \quad \mathsf{maxsave}(c, 2, c) = 0, \quad \mathsf{maxsave}(c, 1, s) = -1$$
$$\mathsf{maxsave}(d, 1, s) = 1, \quad \mathsf{maxsave}(d, 2, c) = -1,$$

So, we next replace the digram $e := (h, 2, c)$ and obtain the system:

$$e(x, y, z) \to d(s(y), x, z), \qquad d(s(x), c(s(0), y), z) \to e(y, s(0), c(x, z))$$

At this point, $f := (d, 1, s)$ is the only diagram with a strictly positive maxsave-value, namely 1. Hence, we replace this digram and get the final compressed system

$$e(x, y, z) \to f(y, x, z), \qquad f(x, c(s(0), y), z) \to e(y, s(0), c(x, z)).$$

Our implementation of MCTreeRePair follows the same strategy as TreeRePair described in the previous section. In a first step we compute for each node p of a tree t_j in the input term list $\bar{t}$ the number $|\operatorname{Var}(t_j|_p)|$ of different variables in the subtree rooted at p. These numbers are necessary to compute the savings of a set of digram occurrences according to (5). Then, as for standard TreeRePair, we insert for every digram d all occurrences from the set $\mathsf{maxOcc}(d, \bar{t})$ into a doubly linked list (one for each digram). Thereby, we also compute the savings $\mathsf{maxsave}(d, \bar{t}) = \mathsf{save}(\mathsf{maxOcc}(d, \bar{t}), \bar{t})$ according to (5). Note that when we replace a certain occurrence (j, p) of the digram $d = [f, i, g]$ in the tree t_j, and the resulting tree is t'_j (i.e., we apply the inverse rule $\mathsf{rule}(d)^{-1}$ at position p in t_j), then we do not have to recompute the numbers $|\operatorname{Var}(t'_j|_q)|$ (for all nodes q of t'_j): In our implementation of the digram replacement, we remove the node pi from the pointer structure representing t_j, The new parent node of pik $(1 \le k \le \operatorname{rk}(g))$ becomes node p. Thereby, the number of different variables below a certain node does not change.

One might try to reduce the computation of the $\mathsf{maxsave}(d, \bar{t})$-values to ordinary digram counting (as done in TreeRePair) as follows: Let t be a term, let u be a subterm of t and let x be a variable. We say that a subterm $s \trianglelefteq t$ is a *maximal non-x subterm of t* if x does not occur in s, but x occurs in every subterm u with $s \triangleleft u$. We denote by t_x the term in which all maximal non-x subterms have been replaced by some dummy symbol N.

▶ **Example 16.** Let $t = f(g(x), h(x, s(y)))$. Then $t_x = f(g(x), h(x, N), t_y = f(N, h(N, s(y))$.

For a term t, let $\{x_1, \ldots, x_n\}$ be the set of variables occurring in t. Then one can check that $\mathsf{maxsave}(d, t)$ equals the number of non-overlapping occurrences of d in the term list $(t_{x_1}, \ldots, t_{x_n})$ minus the cost of d. However this can lead to a quadratic blowup of the input terms. To see that, consider the term

$$t_n = \underbrace{f(\ldots(f}_{n \text{ times}}\, g\, \underbrace{(x_1, \ldots, x_n)}_{n \text{ times}})).$$

where f has rank 1 and g has rank n. This tree has size $2n+1$, but the term list $(t_{x_1}, \ldots, t_{x_n})$ has total size $n(2n + 1)$.

6 Compression and the dependency pairs transformation

The dependency pairs transformation [1] converts the full termination problem of R over Σ into the top termination problem of $\mathrm{DP}(R)$ relative to R, over signature $\Sigma \cup \Sigma^{\#}$, where

$$\mathrm{DP}(R) := \{l^{\#} \to s^{\#} \mid (l \to r) \in R, s = (D, \lambda) \trianglelefteq r, \lambda(\varepsilon) \in \operatorname{defined}(R), s \not\trianglelefteq l\},$$

where $\operatorname{defined}(R)$ is the set of all symbols that occur in root positions of left-hand sides of rules of R, and the operation of *marking* the top symbol is $f(t_1, \ldots, t_n)^{\#} = f^{\#}(t_1, \ldots, t_n)$.

▶ **Example 17.** For (the string rewrite system) $R = \{a^2b^2 \to b^3a^3\}$, we obtain $\operatorname{defined}(R) = \{a\}$ and $\mathrm{DP}(R) = \{a^{\#}ab^2 \to a^{\#}a^2, a^{\#}ab^2 \to a^{\#}a, a^{\#}ab^2 \to a^{\#}\}$.

When using monotone matrix interpretations to prove relative top termination, one uses two-sorted interpretations (called *weakly monotone algebras* [6]). An m-ary marked symbol $f^\#$ is interpreted by a linear function $[f^\#] : (S^n)^m \to S$, whereas an m-ary unmarked symbol f is interpreted by a linear function $[f] : (S^n)^m \to S^n$.

We now discuss how two-sortedness affects compression. We observe that the cost model that just counts matrix multiplications is wrong for computing interpretations for $\mathrm{DP}(R)$, and consequently MCTreeRePair produces inefficient results. This is shown in the following example.

▶ **Example 18.** Let $f^\#$ be a marked binary symbol and g, h be unmarked unary symbols. The interpretation of $f^\#$ has coefficient matrices $F_1^\#, F_2^\#$ (of dimension $1 \times n$), and the interpretation of g (resp., h) has a coefficient matrix G_1 (resp., H_1) of dimension $n \times n$. Consider the computation of the coefficient for variable x in the term $t = f^\#(g(h(x)), g(h(x)))$. It is tempting to first replace the two occurrences of the digram $e = (g, 1, h)$. So we compute $E_1 = G_1 H_1$ (a product of two $(n \times n)$-matrices) with n^3 elementary multiplications. Then we compute the coefficient of x in t as $F_1^\# E_1 + F_2^\# E_1$, which needs another $2n^2$ elementary multiplications (this can be reduced to n^2 multiplications by computing $(F_1^\# + F_2^\#)E_1$). Hence, in total we need $n^3 + 2n^2$ (or $n^3 + n^2$ if we use the alternative) multiplications.

But there is a better way: Compute $(F_1^\# G_1)H_1$, that is, first multiply $F_1^\#$ by G_1 and then the result by H_1, and similarly $(F_2^\# G_1)H_1$, which needs in total only $4n^2$ multiplications. In terms of digrams, this means that we replace the following digrams (which occur only once) in this order: $c := (f^\#, 1, g)$, $d := (c, 1, h)$, $e := (d, 2, g)$, $f := (e, 2, h)$.

The example shows that computation of the interpretation of marked terms is best done "from the top". In this way we can avoid multiplications of two $(n \times n)$-matrices. We draw the conclusion that we should compress only R, because that is where the expensive multiplications take place.

We now describe how we obtain compression for $\mathrm{DP}(R)$ as a side effect. The reason is that all terms in $\mathrm{DP}(R)$ have shape $f^\#(t_1, \ldots, t_n)$ where each t_i is a subterm of some term in R. This implies that we can extract a compressed version of t_i.

We compress R over Σ, obtaining a compressed system R_c over the extended alphabet Σ_c consisting of Σ and digrams. We then compute the compressed version of $\mathrm{DP}(R)$ by applying a modified operation DP_c on the compressed system R_c. This operation $\mathrm{DP}_c(R_c)$ has two ingredients:

- computation of the set of all subterms (in compressed form) of a compressed term, and
- marking of the top symbol of a compressed term.

In both cases the output term(s) should be compressed, and be obtained without completely unpacking the input term. These operations can be realized in a straightforward manner by expanding digrams as needed, at the current position. We make no attempts at constructing fresh digrams.

▶ **Example 19.** Let f be a unary and $t = f^8(x)$. Consider the compressed term $t_c = d_3(x)$ with digrams $d_3 = (d_2, 1, d_2), d_2 = (d_1, 1, d_1), d_1 = (f, 1, f)$. The proper subterms of t in compressed form are $fd_1d_2(x)$, $d_1d_2(x)$, $fd_2(x)$, $d_2(x)$, $fd_1(x)$, $d_1(x)$, $f(x)$, x, and $d_3(x)^\#$ is $f^\# fd_1d_2(x)$.

▶ **Example 20** (Example 17 continued)**.** For $R = \{a^2b^2 \to b^3a^3\}$ compression produces $R_c = \{de \to ebda\}$ with digrams $d = (a, 1, a), e = (b, 1, b)$. We obtain $\mathrm{DP}_c(R_c) = \{a^\# ae \to a^\# a^2, a^\# ae \to a^\# a, a^\# ae \to a^\#\}$. Note that the right-hand side $a^\# a^2$ of the first rule gets $a^\# a$ from marking the expansion of the digram d, plus an adjacent a.

To compute efficiently the interpretations of marked terms, like $f^\# f d_1 d_2(x)$ from Example 19, we work from the top, i.e., left-to-right. This can be modelled by the introduction of digrams $e_1 = (f^\#, 1, f)$, $e_2 = (e_1, 1, d_1)$, $e_3 = (e_2, 1, d_2)$. The interpretations of these digrams are linear functions from S^n to S. For the computation of the coefficients of these linear functions, we have to multiply a $(1 \times n)$-matrix with a $(n \times n)$-matrix (but never two $(n \times n)$-matrices). We therefore compress $\mathrm{DP}_c(R_c)$ by repeatedly replacing digrams that occur at the root of some term from $\mathrm{DP}_c(R_c)$. The algorithm stops when all children of all top symbols are variables.

We summarize the DP-MCTreeRePair algorithm:

input: a rewriting System R over Σ
$R_c :=$ MCTreeRePair(R)
$D_c := \mathrm{DP}_c(R_c)$
while there exists a digram d that occurs at the top of some rule in D_c
  $D_c :=$ replace each occurence of d in lhs and rhs of D_c
endwhile
output: (D_c, R_c) such that expand$(D_c) = \mathrm{DP}(R)$ and expand$(R_c) = R$.

▶ **Example 21** (Example 20 continued)**.** In $\mathrm{DP}_c(R_c)$ we introduce digrams $d_1 = (a^\#, 1, a), d_2 = (d_1, 1, e), d_3 = (d_1, 1, a)$, and obtain $\{d_2 \to d_3, d_2 \to d_1, d_2 \to a^\#\}$.

We now compare the number of matrix and vector operations for different compression methods applied with the dependency pairs transformation. We compare to a "naive" compression method as well.

▶ **Example 22.** For the symbolic evaluation of an n-dimensional matrix interpretation for the rewriting system from Example 2, Table 1 contains in column (p, q, r) the number of multiplications of a $(p \times q)$-matrix by a $(q \times r)$-matrix.

Table 1 Number of matrix multiplications for the rewriting system from Example 2.

method	$(1, n, 1)$	$(1, n, n)$	$(n, n, 1)$	(n, n, n)
uncompressed $(\mathrm{DP}(R) \cup R)$	4	8	20	18
MCTreeRePair$(\mathrm{DP}(R) \cup R)$	4	8	13	12
DP-MCTreeRePair(R)	9	11	9	8

By applying algorithm DP-MCTreeRePair, the number of matrix-by-matrix multiplications is lowest—in fact it is equal to the number of matrix-by-matrix multiplications of MCTreeRePair(R).

7 Experiments

We implemented a version of MCTreeRePair as described in Sections 5 and 6, and we evaluated our implementation in two settings:

- We evaluated how compression reduces the size of constraint systems for rewrite systems from the Termination Problems Data Base (more precisely, the SRS/TRS standard/relative subsets of TPDB version 8), which consists of 3027 files.
- We determined the influence of compression on the power of an actual termination prover.

The source code is available from `https://github.com/jwaldmann/matchbox`. The complete experimental data (log files) is available from `http://www.informatik.uni-leipzig.de/~noeth/`

To measure the "compressibility" of TPDB problems, we used the matrix multiplication cost from (2) as well as the actual size of the resulting SAT constraint system with fixed parameters for the matrix dimension and the bit width of matrix entries. We compared these measures for the settings with and without compression, for both the original systems and for their DP-transformed versions. Table 2 shows the results. Column "cost" shows the accumulated costs of all terms from the corpus. Column "CNF-size" shows the accumulated number of variables and clauses that are being generated by the "bit blasting" translation from the (arctic) integer constraint problem. For "no compression" and "compression" we use (3×3)-matrices with 3-bit entries, for "DP" and "DP and compression" we use (3×3)-matrices with 5-bit entries. Note that we obtain an overall compression ratio of about 3 for both the matrix multiplication cost and the actual CNF size (number of clauses). For DP, these ratios are 3.44 and 1.71, respectively. We conclude that our cost model gives, on average, a very good approximation of the real cost.

Table 2 Total cost and CNF-size with and without compression, for 3027 systems from TPDB.

method	cost	CNF-size (variables, clauses)
no compression	$1.61 \cdot 10^6$	$4.04 \cdot 10^8, 3.23 \cdot 10^9$
compression	$5.18 \cdot 10^5$	$1.30 \cdot 10^8, 1.04 \cdot 10^9$
dependency pairs (DP)	$1.51 \cdot 10^6$	$1.92 \cdot 10^9, 6.22 \cdot 10^9$
DP and compression	$4.39 \cdot 10^5$	$1.11 \cdot 10^9, 3.63 \cdot 10^9$

For estimating the effect of compression on the performance of a termination prover, we used a restricted version of *matchbox*. It optionally applies the dependency pairs transformation and then repeats the following steps until there are no more strict rules:

- If the system is linear, remove rules by additive weights (linear polynomials of slope 1 with absolute coefficients computed by the GLPK solver for linear inequalities).
- For increasing matrix dimensions, try to remove rules by natural matrix interpretations for original systems [6] (solved by binary bit-blasting) and arctic matrix interpretations for DP-transformed systems [10] (solved by unary bit-blasting [3]). In both cases, MINISAT [5] is used as the backend solver.

We apply the "cheap" method (additive weights) first so that the remaining constraint systems are non-trivial. We isolate the effect of compression by using matrix interpretations as the only non-cheap method.

Our experiment then consists of a comparison between the performances of an implementation with and without compression. The following parameters are fixed at the beginning: the boolean encoding of numbers (in particular, their bit width), the matrix dimensions that are being used, the compiler settings, runtime settings, and resources of the execution platform (timeout, memory size, cores). We choose "sensible" values for these parameters, but make no particular attempt to optimize them.

Our implementation exploits parallelism: We search for matrix interpretations in dimensions $1, 2, \ldots, D$ in parallel (for some parameter D that is fixed in advance), i.e., we generate constraint systems $C_1, C_2, \ldots, C_D$ and submit each of them to a separate instance of the SAT solver. As soon as one C_i is solved, we stop the other computations, remove some rules from the input problem (according to the interpretation that was obtained as the solution of C_i), and start afresh. In this way, we actually measure the time that the constraint solver needs in the positive case(s) only. Compare this with a sequential implementation, where we would have to wait for C_i to be recognized as unsolvable, before attempting to solve C_{i+1}. In

this case, the total time would include several unsuccessful attempts as well. But in reality (of proving termination automatically), we are not interested in unsolvable C_i, because they do not contain information on the termination problem. (We cannot distinguish between unsatisfiability due to non-termination, or due to insufficient bit width.)

Table 3 shows the results of our experiments. Column "# yes instances" shows the number of rewrite systems for which termination is successfully proven within 1 minute (the time out). Column "average time yes" is the average time needed to prove termination overall yes instances. It shall be noted that the number of "# yes instances" includes the number of systems for which termination could be proven by "cheap" methods, as described above (there were 50 such cases without using the dependency pairs method, and 250 with it). As can be seen, the number of systems which can be proven to be terminating increases by about 7% (3,5% for DP) when using MCTreeRepair-compression. We conclude that compression of rewriting systems using MCTreeRePair does improve the power of a termination prover that uses a constraint solver to find interpretations.

Table 3 also shows our results when "naive" compression based on TreeRePair (as outlined in Section 4) instead of MCTreeRePair is used in the termination prover. Surprisingly, the number of systems for which termination can be proven is less than without any compression. Here is a possible explanation: When computing the interpretation of a term t without variables bottom-up, only cheap matrix-by-vector multiplications are needed; expensive matrix-by-matrix multiplications do not occur. But compression based on ordinary TreeRePair only tries to reduce the size of t and therefore may introduce diagrams which lead to expensive matrix-by-matrix multiplications when evaluating the digrams. On the other hand, MCTreeRePair will not introduce any diagrams in t: Every digramm occurrence $d = [f, i, g]$ in t, where g has at least arity 1 yields the negative contribution $-\operatorname{cost}(d) = -\operatorname{rk}(g)$ to the savings according to (5).

Table 3 Influence of compression on the matchbox termination prover.

method	average time yes	# yes instances
no compression	11.9	584
compression with MCTreeRePair	12.2	628
naive compression with TreeRePair	11.9	571
dependency pairs (DP)	1.85	681
DP and compression	4.10	709

All values in Table 3 were obtained for an unlimited maximal rank for diagrams. We also experimented with bounded maximal ranks, and it turned out that the optimal value (w.r.t. resulting number of termination proofs for Matchbox using DP-MCTreeRePair) seems to be $r = 4$ (whis is also the optimal value for XML-compression based on TreeRePair in [12]): The number of proofs is slightly larger than with unbounded rank, and we have no explanation at the moment.

8 Discussion and summary

Does compression really preserve semantics? For any given interpretation of function symbols, the interpretation of a compressed term is equivalent to the interpretation of the original term. The underlying reason is that matrix multiplication is associative: digrams correspond to sub-multiplications.

When solving matrix constraints by bit-blasting, the range for matrix elements is a finite set, prescribed by the bit width of the encoding. This implies that arithmetical operations may overflow (For natural numbers, addition and multiplication may overflow; for arctic numbers, multiplication may), so they are partial functions. These partial functions are no longer associative: For instance, consider the integer product abc for three bit integers a, b and c. For $a = 7$, $b = 7$, $c = 0$ the product $a(bc)$ is representable, while the product $(ab)c$ is not. Now, (ab) could be a digram that occurs during compression, while (bc) could correspond to an evaluation of the uncompressed term. Then the constraint system generated from the compressed terms may be unsatisfiable, while the original system is satisfiable. Take the bit width w as a parameter. It can be shown that the original system O and the compressed system C are equivalent in the sense that for each satisfying assignment s of $O(w)$ there is some $w' \geq w$ such that a padded version s' of s satisfies $C(w')$.

Does compression work with more advanced termination methods? The basic dependency pairs method has many refinements [9, 8], which we ignore here since they appear orthogonal to the topic of compression. For instance, by using (estimated) dependency graphs, one obtains termination subproblems that refer to subsets of $\mathrm{DP}(R)$. The "usable rules" method creates subproblems that contain subsets of R. In both cases, compression can be obtained by the methods shown in Section 6.

Is the data base sufficient? We were running experiments on problems contained in the Termination Problems Data Base (TPDB). It may be argued that most of the problems in TPDB are small, and do not need compression. Our experiments show that even for small problems, compression may help. For instance, consider problem TRS/Gebhardt_06/02.trs. We apply the DP transformation, a simplex solver, and arctic matrix interpretations. The nontrivial part is to find a matrix interpretation of dimension 4. We use 5 bit arctic unary numbers. Without compression, we get 16 multiplications of (4×4)-matrices, resulting in a CNF with 45423 variables and 146770 clauses, which is solved in 18 seconds. With compression, we get only 7 matrix multiplications, the CNF has 22303 variables and 71970 clauses, and is solved in 10 seconds. Another point is that TPDB problems might not be typical "real life" termination problems. It appears that most of the TPDB problems are hand-crafted: They are taken from publications, where they serve to illustrate certain isolated points. So, they tend to be small but hard (and trivial only when one applies a specific, advanced method). Application problems, on the other hand, may be large but "easy", and appear hard only because of their size, and compression can reduce size.

Extensions. The method given in the paper counts matrix-by-matrix multiplications only. Our experiments confirm that this is a reasonable simplification. For still better compression, we additionally need to take into account the cost of vector-by-matrix multiplications at the top of DP-transformed rules, and also the matrix-by-vector multiplications for evaluating absolute parts. This implies extensions in the definition of the savings of a digram, and in the algorithm to incrementally update the savings information. In full generality, this includes the "matrix chain multiplication" optimization problem—and goes beyond it, since it is not just about parenthesizing matrix chains, but also about re-using subexpressions. We leave that as possible direction for future work.

Conclusion. We presented the MCTreeRePair algorithm, which reduces the size of constraint systems that determine matrix interpretations for automatically proving termination of rewriting. MCTreeRePair is based on the tree compression algorithm TreeRePair. To obtain a good compression for these constraint systems, we enriched TreeRePair with a cost function that is sensitive to the number of variables in subtrees. We showed that this addi-

tional information can be handled without too much extra work. We also showed that the dependency pairs transformation can be applied to compressed systems directly. We tested our implementation for problems from the Termination Problems Data Base. MCTreeRe-Pair reduces the sizes of the resulting constraint systems by factor of approx. 1/3. We also provided experimental evidence showing that smaller constraint systems tend to be solved faster by a state-of-the art solver. To conclude, compression seems to be a useful addition to termination provers that use interpretation methods.

References

1 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.

2 M. Charikar, E. Lehman, A. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf.Theory*, 51(7):2554–2576, 2005.

3 Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic semiring constraints. In *Proc. SMT 2012*, pages 87–96. `http://smt2012.loria.fr/SMT2012.pdf`, 2012.

4 Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. SAT-based termination analysis using monotonicity constraints over the integers. *TPLP*, 11(4-5):503–520, 2011.

5 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT 2005*, LNCS 3569, pages 61–75. Springer, 2005.

6 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

7 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT 2007*, LNCS 4501, pages 340–354. Springer, 2007.

8 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3):155–203, 2006.

9 Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In *Proc. RTA 2004*, LNCS 3091, pages 249–268. Springer, 2004.

10 Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

11 Dallas S Lankford. On proving term rewriting systems are noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, 1979.

12 Markus Lohrey, Sebastian Maneth, and Roy Mennicke. Tree structure compression with RePair. In *Proc. DCC 2011*, pages 353–362. IEEE Press, 2011.

13 Peter Schneider-Kamp, Carsten Fuhs, René Thiemann, Jürgen Giesl, Elena Annov, Michael Codish, Aart Middeldorp, and Harald Zankl. Implementing RPO and POLO using SAT. In *Deduction and Decision Procedures*, volume 07401 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2007.

A Variant of Higher-Order Anti-Unification*

Alexander Baumgartner[1], Temur Kutsia[1], Jordi Levy[2], and Mateu Villaret[3]

1 **Research Institute for Symbolic Computation (RISC)**
Johannes Kepler University, Linz, Austria
{abaumgar,kutsia}@risc.jku.at
2 **Artificial Intelligence Research Institute (IIIA)**
Spanish Council for Scientific Research (CSIC), Barcelona, Spain
levy@iiia.csic.es
3 **Departament d'Informàtica i Matemàtica Aplicada (IMA)**
Universitat de Girona (UdG), Girona, Spain
villaret@ima.udg.edu

Abstract

We present a rule-based Huet's style anti-unification algorithm for simply-typed lambda-terms in η-long β-normal form, which computes a least general higher-order pattern generalization. For a pair of arbitrary terms of the same type, such a generalization always exists and is unique modulo α-equivalence and variable renaming. The algorithm computes it in cubic time within linear space. It has been implemented and the code is freely available.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.4.1 Mathematical Logic

Keywords and phrases higher-order anti-unification, higher-order patterns.

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.113

1 Introduction

The anti-unification problem of two terms t_1 and t_2 is concerned with finding their generalization, a term t such that both t_1 and t_2 are instances of t under some substitutions. Interesting generalizations are the least general ones. The purpose of anti-unification algorithms is to compute such least general generalizations (lggs).

For higher-order terms, in general, there is no unique higher-order lgg. Therefore, special classes have been considered for which the uniqueness is guaranteed. One of such classes is formed by higher-order patterns. These are λ-terms where the arguments of free variables are distinct bound variables. They have been introduced by Miller [25] and gained popularity because of an attractive combination of expressive power and computational costs: There are practical unification algorithms [28, 27, 26] that compute most general unifiers whenever they exist. Pfenning gave the first algorithm for higher-order pattern anti-unification in the Calculus of Constructions [28], with the intention of using it for proof generalization.

Since then, there have been several approaches to higher-order anti-unification, designing algorithms in various restricted cases. Motivated by applications in inductive learning, Feng and Muggleton [14] proposed anti-unification in $M\lambda$, which is essentially an extension of

* This research has been partially supported by the projects HeLo (TIN2012-33042) and TASSAT (TIN2010-20967-C04-01), by the Austrian Science Fund (FWF) with the project SToUT (P 24087-N18) and by the Generalitat de Catalunya with the grant AGAUR 2009-SGR-1434.

higher-order patterns by permitting free variables to apply to object terms, not only to bound variables. Object terms may contain constants, free variables, and variables which are bound outside of object terms. The algorithm has been implemented and was used for inductive generalization.

Anti-unification in a restricted version of $\lambda 2$ (a second-order λ-calculus with type variables [4]) has been studied in [23] with applications in analogical programming and analogical theorem proving. The imposed restrictions guarantee uniqueness of the least general generalization. This algorithm as well as the one for higher-order patterns by Pfenning [28] have influenced the generalization algorithm used in the program transformation technique called supercompilation [24].

There are other fragments of higher-order anti-unification, motivated by analogical reasoning. A restricted version of second-order generalization developed in [15] has an application in the replay of program derivations. A symbolic analogy model, called Heuristic-Driven Theory Projection, uses yet another restriction of higher-order anti-unification to detect analogies between different domains [18].

The last decade has seen a revived interest in anti-unification. The problem has been studied in various theories (e.g., [1, 2, 9, 19]) and from different application points of view (e.g., [3, 8, 18, 23, 31, 22]). A particularly interesting application comes from software code refactoring, to find similar pieces of code, e.g., in Python, Java [6, 7] and Erlang [22] programs. These approaches are based on the first-order anti-unification [29, 30]. To advance the refactoring and clone detection techniques for languages based on λProlog, one needs to employ anti-unification for higher-order terms. This potential application can serve as a motivation to look into the problem of higher-order anti-unification in more detail.

In this paper, we revisit the problem of higher-order anti-unification, permit arbitrary terms as the input and require higher-order patterns in the output, and present an algorithm in the simply-typed setting. The main contributions can be briefly summarized as follows:

1. Designing a rule-based anti-unification algorithm in simply-typed λ-calculus (in Sect. 3). The input of the algorithm are arbitrary terms in η-long β-normal form. The output is a higher-order pattern. The formulation follows Huet's simple and elegant style [17]. The global function for recording disagreements is represented as a store, in the spirit of [1, 2].
2. Proofs of the termination, soundness, and completeness properties of the anti-unification algorithm (in Sect. 4) and its subalgorithm, which computes permuting matchers between patterns (in Sect. 3.2).
3. Complexity analysis (in Sect. 4): The algorithm computes a least general pattern generalization, which always exists and is *unique* modulo α-equivalence, in *cubic* time and requires linear space. As it is done in related work, we assume that symbols and pointers are encoded in constant space, and basic operations on them performed in constant time.
4. Free open-source implementation for both simply-typed and untyped calculi (Sect. 5).

An extended version of this paper appears as the technical report [5].

Related Work

Here we briefly compare our work with the existing results in higher-order anti-unification. The approaches which are closest to us are the following two:

- In [28], Pfenning studied anti-unification in the Calculus of Construction, whose type system is richer than the simple types we consider. Both the input and the output was

required to be higher-order patterns. Some questions have remained open, including the efficiency, applicability, and implementations of the algorithm. Due to the nature of type dependencies in the calculus, the author was not able to formulate the algorithm in Huet's style [17], where a global function is used to guarantee that the same disagreements between the input terms are mapped to the same variable. The complexity has not been studied and the proofs of the algorithm properties have been just sketched.

- Anti-unification in $M\lambda$ [14] is performed on simply-typed terms, where both the input and the output are restricted to a certain extension of higher-order patterns. In this sense it is not comparable to our case, because we do not restrict the input, but require patterns in the output. The paper [14] contains neither the complexity analysis of the $M\lambda$ anti-unification algorithm nor the proofs of its properties.

Some more remotely related / incomparable to us results are listed below:

- Anti-unification studied in [23] is defined in a restricted version of $\lambda 2$. The restriction requires the λ-abstraction not to be used in arguments. The algorithm computes a generalization which is least general with respect to the combination of several orderings defined in the paper. The properties of the algorithm are formally proved, but the complexity has not been analyzed. As the authors point out, the orderings they define are not comparable with the ordering used to compute higher-order pattern generalizations.
- Generalization algorithms in [16] work on second-order terms which contain no λ-abstractions. The output is also restricted: It may contain variables which can be instantiated with multi-hole contexts only. Varying restrictions on the instantiation, various versions of generalizations are obtained. This approach is not comparable with ours.
- The anti-unification algorithm in [18] works on λ-abstraction-free terms as well. It has been developed for analogy making. The application dictates the typical input to be first-order, while their generalizations may contain second-order variables. A certain measure is introduced to compare generalizations, and the algorithm computes those which are preferred by this measure. This approach is not comparable with ours either.
- The approach in [15] is also different from what we do. The anti-unification algorithm there works on a restriction of combinator terms and computes their generalizations (in quadratic time). It has been used for program derivation.

2 Preliminaries

In higher-order signatures we have *types* constructed from a set of *basic types* (typically δ) using the grammar $\tau ::= \delta \mid \tau \to \tau$, where $\to$ is associative to the right. *Variables* (typically $X, Y, Z, x, y, z, a, b, \ldots$) and *constants* (typically $f, c, \ldots$) have an assigned type.

λ-*terms* (typically $t, s, u, \ldots$) are built using the grammar

$$t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2$$

where x is a variable and c is a constant, and are typed as usual. Terms of the form $(\ldots(h\ t_1)\ldots t_m)$, where h is a constant or a variable, will be written as $h(t_1, \ldots, t_m)$, and terms of the form $\lambda x_1.\cdots.\lambda x_n.t$ as $\lambda x_1, \ldots, x_n.t$. We use $\vec{x}$ as a short-hand for $x_1, \ldots, x_n$.

Other standard notions of the simply typed λ-calculus, like bound and free occurrences of variables, α-conversion, β-reduction, η-long β-normal form, etc. are defined as usual (see [12]). By default, terms are assumed to be written in η-long β-normal form. Therefore, all terms have the form $\lambda x_1, \ldots, x_n.h(t_1, \ldots, t_m)$, where $n, m \geqslant 0$, h is either a constant or a variable, $t_1, \ldots, t_m$ have also this form, and the term $h(t_1, \ldots, t_m)$ has a basic type.

The set of free variables of a term t is denoted by $\mathrm{Vars}(t)$. When we write an equality between two λ-terms, we mean that they are equivalent modulo α, β and η equivalence.

The *depth* of a term t, denoted $\mathrm{Depth}(t)$ is defined recursively as follows: $\mathrm{Depth}(x) = \mathrm{Depth}(c) = 1$, $\mathrm{Depth}(h(t_1, \ldots, t_n)) = 1 + \max_i \mathrm{Depth}(t_i)$, and $\mathrm{Depth}(\lambda x.t) = 1 + \mathrm{Depth}(t)$.

For a term $t = \lambda x_1, \ldots, x_n.h(t_1, \ldots, t_m)$ with $n, m \geqslant 0$, its *head* is defined as $\mathrm{Head}(t) = h$.

Positions in λ-terms are defined with respect to their tree representation in the usual way, as string of integers. For instance, in the term $f(\lambda x.\lambda y.g(\lambda z.h(z, y),\ x), \lambda u.g(u))$, the symbol f stands in the position ϵ (the empty sequence), the occurrence of $\lambda x.$ stands in the position 1, the bound occurrence of y in 1.1.1.1.1.2, the bound occurrence of u in 2.1.1, etc.

The *path to a position* in a λ-term is defined as the sequence of symbols from the root to the node at that position (not including) in the tree representation of the term. For instance, the path to the position 1.1.1.1.1 in $f(\lambda x.\lambda y.g(\lambda z.h(z, y), x), \lambda u.g(u))$ is $f, \lambda x, \lambda y, g, \lambda z$.

A *higher-order pattern* is a λ-term where, when written in η-long β-normal form, all free variable occurrences are applied to lists of pairwise distinct (η-long forms of) bound variables. For instance, $\lambda x.f(X(x), Y)$, $f(c, \lambda x.x)$ and $\lambda x.\lambda y.X(\lambda z.x(z), y)$ are patterns, while $\lambda x.f(X(X(x)), Y)$, $f(X(c), c)$ and $\lambda x.\lambda y.X(x, x)$ are not.

Substitutions are finite sets of pairs $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where X_i and t_i have the same type and the X's are pairwise distinct variables. They can be extended to type preserving functions from terms to terms as usual, avoiding variable capture. The notions of substitution *domain* and *range* are also standard and are denoted, respectively, by Dom and Ran.

We use postfix notation for substitution applications, writing $t\sigma$ instead of $\sigma(t)$. As usual, the application $t\sigma$ affects only the free occurrences of variables from $\mathrm{Dom}(\sigma)$ in t. We write $\vec{x}\sigma$ for $x_1\sigma, \ldots, x_n\sigma$, if $\vec{x} = x_1, \ldots, x_n$. Similarly, for a set of terms S, we define $S\sigma = \{t\sigma \mid t \in S\}$. The *composition* of σ and ϑ is written as juxtaposition $\sigma\vartheta$. Yet another standard operation, *restriction* of a substitution σ to a set of variables S, is denoted by $\sigma|_S$.

A substitution σ_1 is *more general* than σ_2, written $\sigma_1 \preceq \sigma_2$, if there exists ϑ such that $X\sigma_1\vartheta = X\sigma_2$ for all $X \in \mathrm{Dom}(\sigma_1) \cup \mathrm{Dom}(\sigma_2)$. The strict part of this relation is denoted by $\prec$. The relation $\preceq$ is a partial order and generates the equivalence relation which we denote by $\simeq$. We overload $\preceq$ by defining $s \preceq t$ if there exists a substitution σ such that $s\sigma = t$.

A term t is called a *generalization* or an *anti-instance* of two terms t_1 and t_2 if $t \preceq t_1$ and $t \preceq t_2$. It is a *higher-order pattern generalization* if additionally t is a higher-order pattern. It is the *least general generalization,* (lgg in short), aka a *most specific anti-instance,* of t_1 and t_2, if there is no generalization s of t_1 and t_2 which satisfies $t \prec s$.

An *anti-unification problem* (shortly AUP) is a triple $X(\vec{x}) : t \triangleq s$ where

- $\lambda\vec{x}.X(\vec{x})$, $\lambda\vec{x}.t$, and $\lambda\vec{x}.s$ are terms of the same type,
- t and s are in η-long β-normal form, and
- X does not occur in t and s.

The variable X is called a *generalization variable.* The term $X(\vec{x})$ is called the *generalization term.* The variables that belong to $\vec{x}$, as well as bound variables, are written in the lower case letters $x, y, z, \ldots$. Originally free variables, including the generalization variables, are written with the capital letters $X, Y, Z, \ldots$. This notation intuitively corresponds to the usual convention about syntactically distinguishing bound and free variables.

An *anti-unifier* of an AUP $X(\vec{x}) : t \triangleq s$ is a substitution σ such that $\mathrm{Dom}(\sigma) = \{X\}$ and $\lambda\vec{x}.X(\vec{x})\sigma$ is a term which generalizes both $\lambda\vec{x}.t$ and $\lambda\vec{x}.s$.

An anti-unifier of $X(\vec{x}) : t \triangleq s$ is *least general* (or *most specific*) if there is no anti-unifier ϑ of the same problem that satisfies $\sigma \prec \vartheta$. Obviously, if σ is a least general anti-unifier of an AUP $X(\vec{x}) : t \triangleq s$, then $\lambda\vec{x}.X(\vec{x})\sigma$ is an lgg of $\lambda\vec{x}.t$ and $\lambda\vec{x}.s$.

Here we consider a variant of higher-order anti-unification problem:

Given: Higher-order terms t and s of the same type in η-long β-normal form.
Find: A higher-order pattern generalization r of t and s.

The problem statement means that we are looking for r which is least general among all higher-order patterns which generalize t and s. There can still exist a term which is less general than r, generalizes both s and t, but is not a higher-order pattern. For instance, if $t = \lambda x, y.f(h(x, x, y), h(x, y, y))$ and $s = \lambda x, y.f(g(x, x, y), g(x, y, y))$, then $r = \lambda x, y.f(Y_1(x, y), Y_2(x, y))$ is a higher-order pattern, which is an lgg of t and s. However, the term $\lambda x, y.f(Z(x, x, y), Z(x, y, y))$, which is not a higher-order pattern, is less general than r and generalizes t and s.

Below we assume that in the AUPs of the form $X(\vec{x}) : t \triangleq s$, the term $\lambda\vec{x}.X(\vec{x})$ is a higher-order pattern.

3 The Algorithm the Higher-Order Anti-Unification Variant

3.1 The Rules

The higher-order anti-unification algorithm is formulated in a rule-based manner working on triples $A; S; \sigma$ (*systems*). Here A is a set of AUPs of the form $\{X_1(\vec{x_1}) : t_1 \triangleq s_1, \ldots, X_n(\vec{x_n}) : t_n \triangleq s_n\}$ where each X_i occurs in $A \cup S$ only once, S is a set of already solved AUPs (the store), and σ is a substitution (computed so far) mapping variables to patterns.

▸ Remark. One assumption we make on the set $A \cup S$ is that each occurrence of λ binds a distinct name variable (in other words, all names of bound variables are distinct).

Dec: **Decomposition**

$$\{X(\vec{x}) : h(t_1, \ldots, t_m) \triangleq h(s_1, \ldots, s_m)\} \cup A;\ S;\ \sigma \Longrightarrow \{Y_1(\vec{x}) : t_1 \triangleq s_1, \ldots, Y_m(\vec{x}) : t_m \triangleq s_m\} \cup A;\ S;\ \sigma\{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\},$$

where h is a constant or $h \in \vec{x}$, and $Y_1, \ldots, Y_n$ are fresh variables of the corresponding types.

Abs: **Abstraction**

$$\{X(\vec{x}) : \lambda y.t \triangleq \lambda z.s\} \cup A;\ S;\ \sigma \Longrightarrow \{X'(\vec{x}, y) : t \triangleq s\{z \mapsto y\}\} \cup A;\ S;\ \sigma\{X \mapsto \lambda\vec{x}, y.X'(\vec{x}, y)\}.$$

where X' is a fresh variable of the appropriate type.

Sol: **Solve**

$$\{X(\vec{x}) : t \triangleq s\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{Y(\vec{y}) : t \triangleq s\} \cup S;\ \sigma\{X \mapsto \lambda\vec{x}.Y(\vec{y})\},$$

where t and s are of a basic type, $\mathrm{Head}(t) \neq \mathrm{Head}(s)$ or $\mathrm{Head}(t) = \mathrm{Head}(s) = Z \notin \vec{x}$, $\vec{y}$ is a subsequence of $\vec{x}$ consisting of the variables that appear freely in t or in s, and Y is a fresh variable of the corresponding type.

Mer: **Merge**

$$A;\ \{X(\vec{x}) : t_1 \triangleq t_2, Y(\vec{y}) : s_1 \triangleq s_2\} \cup S;\ \sigma \Longrightarrow A;\ \{X(\vec{x}) : t_1 \triangleq t_2\} \cup S;\ \sigma\{Y \mapsto \lambda\vec{y}.X(\vec{x}\pi)\},$$

where $\pi : \{\vec{x}\} \to \{\vec{y}\}$ is a bijection, extended as a substitution, with $t_1\pi = s_1$ and $t_2\pi = s_2$.

One can easily show that a triple obtained from $A; S; \sigma$ by applying any of the rules above to a system is indeed a system: For each expression $X(\vec{x}) : t \triangleq s \in A \cup S$, the terms $X(\vec{x})$, t and s have the same type, $\lambda\vec{x}.X(\vec{x})$ is a higher-order pattern, s and t are in η-long

β-normal form, and X does not occur in t and s. Moreover, all generalization variables are distinct and substitutions map variables to patterns.

The property that each occurrence of λ in $A \cup S$ binds a unique variable is also maintained. It guarantees that in the Abs rule, the variable y is fresh for s. After the application of the rule, y will appear nowhere else in $A \cup S$ except $X'(\vec{x}, y)$ and, maybe, t and s.

Like in the anti-unification algorithms working on triple systems [1, 2, 19], the idea of the store here is to keep track of already solved AUPs in order to reuse in generalizations an existing variable. This is important, since we aim at computing lggs.

The Mer rule requires solving a matching problem $\{t_1 \Rightarrow s_1, t_2 \Rightarrow s_2\}$ with the substitution π which bijectively maps the variables from $\vec{x}$ to the variables from $\vec{y}$. In general, when we want to find a solution of a matching problem P, which bijectively maps variables from a finite set D to a finite set R, we say that we are looking for a *permuting matcher* of P from D to R. The sets D and R are supposed to have the same cardinality.

Note that a permuted matcher, if it exists, is unique. It follows from the fact that there can be only one capture-avoiding renaming of free variables which matches a higher-order term to another. Since P is a matching problem for higher-order terms with free variables from D and their potential values from R, it can have at most one such matcher. By $\mathsf{match}(D, R, P)$, we denote such a permuting matcher of P from D to R, when it exists. Otherwise, $\mathsf{match}(D, R, P) = \bot$. An algorithm that computes it is given in Sect. 3.2 below.

To compute generalizations for terms t and s, we start with $\{X : t \triangleq s\}; \varnothing; \varnothing$, where X is a fresh variable, and apply the rules as long as possible. We denote this procedure by $\mathfrak{P}$, to indicate that we compute patterns. The system to which no rule applies has the form $\varnothing; S; \varphi$, where Mer does not apply to S. We call it the final system. When $\mathfrak{P}$ transforms $\{X : t \triangleq s\}; \varnothing; \varnothing$ into a final system $\varnothing; S; \varphi$, we say that *result computed* by $\mathfrak{P}$ is $X\varphi$.

▸ **Example 3.1.** A couple of examples illustrating the generalizations computed by $\mathfrak{P}$:

- Let $t = \lambda x, y.f(U(g(x), y), U(g(y), x))$ and $s = \lambda x', y'.f(h(y', g(x')),\ h(x', g(y')))$. Then $\mathfrak{P}$ performs the following transformations:

$$
\begin{aligned}
& \{X : \lambda x, y.f(U(g(x), y), U(g(y), x)) \triangleq \lambda x', y'.f(h(y', g(x')), h(x', g(y')))\}; \varnothing; \varnothing \\
\Longrightarrow^{2}_{\mathsf{Abs}}\ & \{X'(x, y) : f(U(g(x), y), U(g(y), x)) \triangleq f(h(y, g(x)), h(x, g(y)))\}; \varnothing; \\
& \{X \mapsto \lambda x, y.X'(x, y)\} \\
\Longrightarrow_{\mathsf{Dec}}\ & \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x)), Y_2(x, y) : U(g(y), x) \triangleq h(x, g(y))\}; \varnothing; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \{Y_2(x, y) : U(g(y), x) \triangleq h(x, g(y))\}; \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x))\}; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Sol}}\ & \varnothing; \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x)), Y_2(x, y) : U(g(y), x) \triangleq h(x, g(y))\}; \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_2(x, y)), \dots\} \\
\Longrightarrow_{\mathsf{Mer}}\ & \varnothing; \{Y_1(x, y) : U(g(x), y) \triangleq h(y, g(x))\} \\
& \{X \mapsto \lambda x, y.f(Y_1(x, y), Y_1(y, x)), \dots, Y_2 \mapsto \lambda x, y.Y_1(y, x)\}
\end{aligned}
$$

  The computed result is $r = \lambda x, y.f(Y_1(x, y), Y_1(y, x))$. It generalizes the input terms t and s: $r\{Y_1 \mapsto \lambda x, y.U(g(x), y)\} = t$ and $r\{Y_1 \mapsto \lambda x, y.h(y, g(x))\} = s$. These substitutions can be read from the final store.
- For $\lambda x, y, z.g(f(x, z), f(y, z), f(y, x))$ and $\lambda x', y', z'.g(h(y', x'), h(x', y'), h(z', y'))$, $\mathfrak{P}$ computes their generalization $\lambda x, y, z.f(Y_1(x, y, z), Y_1(y, x, z), Y_1(y, z, x))$

- For $\lambda x, y.f(\lambda z.U(z, y, x), U(x, y, x))$ and $\lambda x', y'.f(\lambda z'.h(y', z', x'),\ h(y', x', x'))$, $\mathfrak{P}$ computes their generalization $\lambda x, y.f(\lambda z.Y_1(x, y, z), Y_2(x, y))$.

As one can see, the computed results are higher-order pattern generalizations of the input terms. Below we will prove it formally, when we establish soundness of $\mathfrak{P}$. The computed results are, in fact, pattern lggs. The Completeness Theorem in the Section 4 states this.

From the examples one can notice yet another advantage of using the store (besides helping in the merging): In the final system, it contains AUPs from which one can get the substitutions that show how the original terms can be obtained from the computed result.

3.2 Computation of Permuting Matchers

In this section we describe the algorithm $\mathfrak{M}$ to compute permuting matchers. It is a rule-based algorithm working on quintuples of the form $D;\ R;\ P;\ \rho;\ \pi$ (also called systems) where D is a set of domain variables, R is a set of range variables, D and R have the same cardinality and are disjoint, P is a set of matching problems of the form $\{s_1 \Rightarrow t_1, \ldots, s_m \Rightarrow t_m\}$, and ρ and π are substitutions (computed so far) mapping variables to variables. Here ρ is supposed to keep bound variable renamings to deal with abstractions, while in π we compute the permuting matcher to be returned in case of success. The rules are the following:

Dec-M: **Decomposition**

$$D;\ R;\ \{h_1(t_1, \ldots, t_m) \Rightarrow h_2(s_1, \ldots, s_m)\} \cup P;\ \rho;\ \pi \Longrightarrow$$
$$D;\ R;\ \{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\} \cup P;\ \rho;\ \pi,$$

where each of h_1 and h_2 is a constant or a variable, and $h_1 \notin D$ or $h_2 \notin R$, and $h_1\pi = h_2\rho$. These conditions make this rule disjoint from the Per-M rule.

Abs-M: **Abstraction**

$$D;\ R;\ \{\lambda x.t \Rightarrow \lambda y.s\} \cup P;\ \rho;\ \pi \Longrightarrow D;\ R;\ \{t \Rightarrow s\} \cup P;\ \rho\{y \mapsto x\};\ \pi.$$

Per-M: **Permuting**

$$\{x\} \cup D;\ \{y\} \cup R;\ \{x(t_1, \ldots, t_m) \Rightarrow y(s_1, \ldots, s_m)\} \cup P;\ \rho;\ \pi \Longrightarrow$$
$$D;\ R;\ \{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\} \cup P;\ \rho;\ \pi\{x \mapsto y\},$$

where x and y have the same type.

Like in the rules for anti-unification above, also here each occurrence of λ binds a unique variable. The input for $\mathfrak{M}$ is initialized in the Mer rule, which needs to compute $\mathsf{match}(D, R, \{t_1 \Rightarrow s_1, t_2 \Rightarrow s_2\})$. The algorithm has the following steps:

1. Domain/range separation: To make sure that they do not share elements, we rename the domain variables with fresh ones, if necessary. It is not a restriction: If ν is such a renaming substitution, then μ is a permuting matcher of $\{s_1\nu \Rightarrow t_1, s_2\nu \Rightarrow t_2\}$ from $D\nu$ to R iff $(\nu\mu)|_D$ is a permuting matcher of $\{s_1 \Rightarrow t_1, s_2 \Rightarrow t_2\}$ from D to R.
2. Next, we create the initial system $D\nu;\ R;\ \{s_1\nu \Rightarrow t_1, s_2\nu \Rightarrow t_2\};\ \varnothing;\ \varnothing$ and apply the rules Dec-M, Abs-M and Per-M exhaustively. If no rule applies to a system $D;\ R;\ P;\ \rho;\ \pi$ with $P \neq \varnothing$, then it is transformed into $\bot$, called the *failure state*. The system $D;\ R;\ \varnothing;\ \rho;\ \pi$ is called the *success state*. No rule applies to it either.
3. When $\mathfrak{M}$ reaches the success state, we say that $\mathfrak{M}$ computes π. From it, we can return the permuting matcher $(\nu\pi)|_D$. When $\mathfrak{M}$ reaches the failure state, we say that it fails.

▸ **Example 3.2.** To compute the permuting matcher of $\{x(y,z) \Rightarrow x(z,y), X(y,\lambda u.u) \Rightarrow X(z,\lambda v.v)\}$ from $\{x,y,z\}$ to $\{x,y,z\}$ by $\mathfrak{M}$, first, we separate the domain and the range with $\nu = \{x \mapsto x', y \mapsto y', z \mapsto z'\}$, obtaining the initial system $\{x',y',z'\};\{x,y,z\};$ $\{x'(y',z') \Rightarrow x(z,y),\ X(y',\lambda u.u) \Rightarrow X(z,\lambda v.v)\};\varnothing;\varnothing$. Applying the rules of $\mathfrak{M}$, we obtain the success state $\varnothing;\varnothing;\varnothing;\{v \mapsto u\};\{x' \mapsto x, y' \mapsto z, z' \mapsto y\}$. Composing ν and the computed substitution we obtain $\{x \mapsto x, y \mapsto z, z \mapsto y\}$, which is the permuting matcher we were looking for.

The algorithm $\mathfrak{M}$ maintains the following invariants: (Justifications can be found in [5].)

Invariant 1: For each tuple D; R; P; $\rho;\pi$ in a derivation performed by $\mathfrak{M}$, the sets D and R are disjoint and have the same number of elements.

Invariant 2: For each tuple D; R; $\{t_1 \Rightarrow s_1, \ldots, t_m \Rightarrow s_m\}$; ρ; π in a derivation performed by $\mathfrak{M}$, $D \subseteq \cup_{i=1}^{m}\mathrm{Vars}(t_i)$ and $R \subseteq \cup_{i=1}^{m}\mathrm{Vars}(s_i)$.

Invariant 3: For each tuple D_i; R_i; P_i; ρ_i; π_i in a derivation performed by $\mathfrak{M}$ starting from D; R; P; ρ; π, the following equalities hold: $D_i \cup \mathrm{Dom}(\pi_i) = D$ and $R_i \cup \mathrm{Ran}(\pi_i) = R$.

▸ **Theorem 3.3.** *$\mathfrak{M}$ is terminating, sound, and complete.*

Proof. *Termination.* Termination of $\mathfrak{M}$ is straightforward: Each rule strictly reduces the multiset of sizes of matching problems in the tuples it operates on. Since each tuple D; R; P; ρ; π with $P \neq \varnothing$ can be transformed by one of the rules or leads to failure, the final state in the derivation is either the success or the failure state.

Soundness. Soundness of $\mathfrak{M}$ means that if for a given tuple D; R; P; $\varnothing$; $\varnothing$ it computes a substitution π, then π is a permuting matcher of P from D to R. Obviously, π maps variables from D to R. It follows from the way how the Per-M rule constructs π. The fact that π is a matcher is straightforward: $\mathrm{Dom}(\pi) \cap \mathrm{Ran}(\pi) = \varnothing$, the differences between t and s for $t \Rightarrow s \in P$ are either repaired by the bindings from π constructed by Per-M, or the differences are α-equivalences repaired by the bindings from ρ constructed by Abs-M, or the failure occurs since no rule can be applied. The bijection property is more involved: The Per-M rule (namely, the fact that it removes x and y from D and R) and the first invariant guarantee that there is an injective mapping from a subset of D onto a subset of R. Since all variables of D (resp. R) appear freely in the left (resp. right) hand sides of equations in P (the second invariant), each derivation either stops with failure, or eventually reduces D and R to $\varnothing$ by applications of Per-M (see the first invariant, the same number of elements in D and R). The latter, by the third invariant, means that there is an injective mapping from D onto R, expressed by π. Hence, π is a bijection from D to R and $\mathfrak{M}$ is sound.

Completeness. Recall that for each D, R, and P, if there exists a permuting matcher of P from D to R, then it is unique. Since we have already proved soundness of $\mathfrak{M}$, we have only to show that if there exists a permuting matcher of P from D to R, then $\mathfrak{M}$ does not fail for D; R; P; $\varnothing$; $\varnothing$. Let μ be such a matcher. Then $t\mu = s$ for all $t \Rightarrow s \in P$. This means that, if t has a form $h_1(t_1,\ldots,t_n)$, then s should be $h_2(s_1,\ldots,s_n)$ and $h_1\mu = h_2$, $t_i\mu = s_i$ for all $1 \leqslant i \leqslant n$. If t has a form $\lambda x.t'$, then s should be of the form $\lambda y.s'$ and $t'\mu = s'\{y \mapsto x\}$.

Assume by contradiction that $\mathfrak{M}$ fails. That means that there exists the system D_k; R_k; $\{t \Rightarrow s\} \cup P_k$; ρ_k; π_k to which no rule applies. Since the steps performed by $\mathfrak{M}$ before it either decompose the terms argumentwise (Dec-M and Per-M), or remove abstraction (Abs-M), by the definitions of matcher and substitution application we should have $t\mu = s\rho_k$. This equation means that t and s have the same types. Hence, the only case why no rule in $\mathfrak{M}$ applies to the system is that t and s should be, respectively, of the form $h_1(t_1,\ldots,t_n)$

and $h_2(s_1, \ldots, s_m)$ with $h_1\pi_k \neq h_2\rho_k$, where $h_1 \notin D_k$ or $h_2 \notin R_k$. Because of the uniqueness of the matcher, $\pi_k = \mu|_{D\setminus D_k}$. On the other hand, $h_1\mu = h_2\rho_k$, because μ matches t to $s\rho_k$.

Hence, we have $h_1\mu|_{D\setminus D_k} \neq h_2\rho_k$ where $h_1 \notin D_k$ or $h_2 \notin R_k$, and $h_1\mu = h_2\rho_k$. The latter means that either $h_1 \in D$ and $h_2 \in R$, or $h_1 \notin D$ and $h_2 \notin R$, because D and R are disjoint, the permuting matcher μ bijectively maps D to R, and ρ_k does not affect R.

Case 1: $h_1 \in D$ and $h_2 \in R$. Because of $h_1\mu|_{D\setminus D_k} \neq h_2\rho_k$, we have $h_1 \in D_k$. If $h_2 \notin R_k$, then there exists some $x \in D$, such that $x \neq h_1$ and $x\mu = h_2$, which contradicts the fact that μ is injective. If $h_2 \in R_k$, we get a contradiction with the condition $h_1 \notin D_k$ or $h_2 \notin R_k$. Hence, the case with $h_1 \in D$ and $h_2 \in R$ is impossible.

Case 2: $h_1 \notin D$ and $h_2 \notin R$. Then $h_1 = h_2\rho_k$ should hold, because $h_1\mu = h_2\rho_k$ and $h_1 \notin \mathrm{Dom}(\mu) = D$. We again get the contradiction, this time with $h_1\mu|_{D\setminus D_k} \neq h_2\rho_k$.

The obtained contradictions show that if there exists a permuting matcher of P from D to R, then $\mathfrak{M}$ does not fail for D; R; P; $\varnothing$; $\varnothing$, which implies completeness of $\mathfrak{M}$. ◂

▸ **Theorem 3.4.** *The algorithm $\mathfrak{M}$ has linear space and time complexity.*

Proof. For the input consisting of the sets of domain variables D, range variables R, and matching equations P, the size is the cardinality of $D \cup R$ plus the number of symbols in P.

The terms to be matched can be represented as trees in the standard way. The sets D and R can be encoded as hash tables. These representations occupy space linear to the size of the input. The space can grow at most twice by representing renaming and permuting substitutions as hash tables. Hence, the space complexity is linear.

As for the time complexity, we can see that the algorithm visits each node of the trees to be matched at most once. At the initial step, renaming all variables in D with fresh ones can take only linear time with the help of the hash table for the renaming substitution.

After that, we perform the following linear time steps: Collecting the set of bound variables V_r appearing in the right sides of matching equations in P, constructing the initial hash tables T_D and T_R for (the renamed) D and R (we can assume that the hash functions are perfect), and constructing two hash tables for substitutions. The one for permuting substitutions is denoted by T_π. Its set of keys is D. We can reuse the same hash function as for T_D. Each address in T_π is initialized with null. Another table, T_ρ, is designed for renaming substitutions. Its set of keys is V_r. We assume a perfect hash function also here.

The operations performed at each node are the following ones: (Note that the substitution compositions in the rules, due to the disjointness of D and R, amounts to only adding a new pair to the existing substitution.)

By Dec-M: First, look up the value for h_1 in T_D, to make sure that $h_1 \notin D$. If D contains the entry for h_1, then look up the value for h_2 in T_R, to make sure that $h_2 \notin R$. If the latter test fails, the rule is not applicable.

Next, if either $h_1 \notin D$ or $h_2 \notin R$, then look up the value for h_1 in T_π, look up the value for h_2 in T_ρ, and compare them with each other. If the values of h_1 or h_2 are not found in the tables, then just use the corresponding h (i.e., h_1 or h_2) in the comparison.

By Abs-M: Modifying an entry in T_ρ: For a renaming substitution $\{y \mapsto x\}$, we put x in the table at the address corresponding to the hash index of y: $T_\rho[hash(y)] = x$. Since all bound variables are distinct, we will not have to modify the same entry in T_ρ again.

By Per-M: Modifying an entry for x in T_π: For a substitution $\{x \mapsto y\}$, we put y in the address corresponding to the hash index of x: $T_\pi[hash(x)] = y$. As we destroy the entries for x in T_D and for y in T_R, we will not modify the same entry again.

All our hash functions are perfect. Searching, insertion and deletion in hash tables with perfect hash functions are done in constant time. We assume that two alphabet symbols can be compared in constant time. Hence, all the operations performed by $\mathfrak{M}$ at each node of the input trees are done in constant time. It implies that $\mathfrak{M}$ has linear time complexity. ◂

4 Properties of the Anti-Unification Algorithm

▸ **Theorem 4.1** (Termination). *The procedure $\mathfrak{P}$, which uses $\mathfrak{M}$ to compute permuting matchers, terminates for all input terms t and s.*

Proof. We define the measure of $A; S; \sigma$ as a pair of multisets $(M(A), M(S))$, where the multiset $M(L) = \{\min(\mathrm{Depth}(t), \mathrm{Depth}(s)) \mid X(\vec{x}) : t \triangleq s \in L\}$ for any L. Measures are compared lexicographically. Obviously, each rule in $\mathfrak{P}$ strictly reduces it. The ordering is well-founded. The procedure $\mathfrak{M}$ in the rule Mer is terminating. Hence, $\mathfrak{P}$ terminates. ◂

▸ **Theorem 4.2** (Soundness). *If $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S; \sigma$ is a derivation in $\mathfrak{P}$, then*

(a) *$X\sigma$ is a higher-order pattern in η-long β-normal form,*
(b) *$X\sigma \preceq t$ and $X\sigma \preceq s$.*

Proof. To prove that $X\sigma$ is a higher-order pattern, we use the facts that first, X is a higher order pattern and, second, at each step $A_1; S_1; \varphi \Longrightarrow A_2; S_2; \varphi\vartheta$ if $X\varphi$ is a higher-order pattern, then $X\varphi\vartheta$ is also a higher-order pattern. The latter property follows from stability of patterns under substitution application and from the fact that substitutions in the rules map variables to higher-order patterns. As for $X\sigma$ being in η-long β-normal form, this is guaranteed by the series of applications of the Abs rule, even if Dec introduces an AUP whose generalization term is not in this form. It finishes the (sketch of the) proof of (4.2).

Proving (4.2) is more involved. First, we prove that if $A_1; S_1; \varphi \Longrightarrow A_2; S_2; \varphi\vartheta$ is one step, then for any $X(\vec{x}) : t \triangleq s \in A_1 \cup S_1$, we have $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$. Note that if $X(\vec{x}) : t \triangleq s$ was not transformed at this step, then this property trivially holds for it. Therefore, we assume that $X(\vec{x}) : t \triangleq s$ is selected and prove the property for each rule:

Dec: Here $t = h(t_1, \ldots, t_m)$, $s = h(s_1, \ldots, s_m)$, and $\vartheta = \{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\}$. Then $X(\vec{x})\vartheta = h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))$. Let ψ_1 and ψ_2 be substitutions defined, respectively, by $Y_i\psi_1 = \lambda\vec{x}.t_i$ and $Y_i\psi_2 = \lambda\vec{x}.s_i$ for all $1 \leqslant i \leqslant m$. Such substitutions obviously exist since the Y's introduced by the Dec rule are fresh. Then $X(\vec{x})\vartheta\psi_1 = h(t_1, \ldots, t_m)$, $X(\vec{x})\vartheta\psi_2 = h(s_1, \ldots, s_m)$ and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

Abs: Here $t = \lambda y_1.t'$, $s = \lambda y_2.s'$, and $\vartheta = \{X \mapsto \lambda\vec{x}, y.X'(\vec{x}, y)\}$. Then $X(\vec{x})\vartheta = \lambda y.X'(\vec{x}, y)$. Let $\psi_1 = \{X' \mapsto \lambda\vec{x}, y.t'\}$ and $\psi_2 = \{X' \mapsto \lambda\vec{x}, y.s'\}$. Then $X(\vec{x})\vartheta\psi_1 = \lambda y.t' = t$, $X(\vec{x})\vartheta\psi_2 = \lambda y.s' = s$, and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

Sol: We have $\vartheta = \{X \mapsto \lambda\vec{x}.Y(\vec{y})\}$, where $\vec{y}$ is the subsequence of $\vec{x}$ consisting of the variables that appear freely in t or s. Let $\psi_1 = \{Y \mapsto \lambda\vec{y}.t\}$ and $\psi_2 = \{Y \mapsto \lambda\vec{y}.s\}$. Then $X(\vec{x})\vartheta\psi_1 = t$, $X(\vec{x})\vartheta\psi_2 = s$, and, hence, $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$.

If Mer applies, then there exists $Y(\vec{y}) : t' \triangleq s' \in S_1$ such that $\mathsf{match}(\{\vec{x}\}, \{\vec{y}\}, t \Rightarrow t', s \Rightarrow s')$ is a permuting matcher π, and $\vartheta = \{Y \mapsto \lambda\vec{y}.X(\vec{x}\pi)\}$. Then $X(\vec{x})\vartheta \preceq t$ and $X(\vec{x})\vartheta \preceq s$ obviously hold. As for the $Y(\vec{y}) : t' \triangleq s'$, let $\psi_1 = \{X \mapsto \lambda\vec{x}.t\}$ and $\psi_2 = \{X \mapsto \lambda\vec{x}.s\}$. Then $Y(\vec{y})\vartheta\psi_1 = (\lambda\vec{x}.t)(\vec{x}\pi) = t\pi = t'$, $Y(\vec{y})\vartheta\psi_2 = (\lambda\vec{x}.s)(\vec{x}\pi) = s\pi = s'$, and, hence, $Y(\vec{y})\vartheta \preceq t'$ and $Y(\vec{y})\vartheta \preceq s'$.

Now, we proceed by induction on the length of derivation l. In fact, we will prove a more general statement: If $A_0; S_0; \vartheta_0 \Longrightarrow^* \varnothing; S_n; \vartheta_0\vartheta_1 \cdots \vartheta_n$ is a derivation in $\mathfrak{P}$, then for any $X(\vec{x}) : t \triangleq s \in A_0 \cup S_0$ we have $X(\vec{x})\vartheta_1 \cdots \vartheta_n \preceq t$ and $X(\vec{x})\vartheta_1 \cdots \vartheta_n \preceq s$.

When $l = 1$, it is exactly the one-step case we just proved. Assume that the statement is true for any derivation of the length n and prove it for a derivation $A_0; S_0; \vartheta_0 \Longrightarrow A_1; S_1; \vartheta_0\vartheta_1 \Longrightarrow^* \varnothing; S_n; \vartheta_0\vartheta_1 \cdots \vartheta_n$ of the length $n+1$.

Below the composition $\vartheta_i\vartheta_{i+1}\cdots\vartheta_k$ is abbreviated as ϑ_i^k with $k \geqslant i$. Let $X(\vec{x}) : t \triangleq s$ be an AUP selected for transformation at the current step. (Again, the property trivially holds for the AUPs which are not selected.) We consider each rule:

Dec: $t = h(t_1, \ldots, t_m)$, $s = h(s_1, \ldots, s_m)$ and $X(\vec{x})\vartheta_1^1 = h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))$. By the induction hypothesis, $Y_i(\vec{x})\vartheta_2^n \preceq t_i$ and $Y_i(\vec{x})\vartheta_2^n \preceq s_i$ for all $1 \leqslant i \leqslant m$. By construction of ϑ_2^n, if there is $U \in \mathrm{Vars}(\mathrm{Ran}(\vartheta_2^n))$, then there is an AUP of the form $U(\vec{u}) : t' \triangleq s' \in S_n$. Let σ (resp. φ) be a substitution which maps each such U to the corresponding t' (resp. s'). Then $Y_i(\vec{x})\vartheta_2^n\sigma = t_i$ and $Y_i(\vec{x})\vartheta_2^n\varphi = s_i$. Since $X(\vec{x})\vartheta_1^n = h(Y_1(\vec{x}), \ldots, Y_m(\vec{x}))\vartheta_2^n$, we get that $X(\vec{x})\vartheta_1^n\sigma = t$, $X(\vec{x})\vartheta_1^n\varphi = s$, and, hence, $X(\vec{x})\vartheta_1^n \preceq t$ and $X(\vec{x})\vartheta_1^n \preceq s$.

Abs: Here $t = \lambda y_1.t'$, $s = \lambda y_2.s'$, $X(\vec{x})\vartheta_1^1 = \lambda y.X'(\vec{x}, y)$, and A_1 contains the AUP $X'(\vec{x}, y) : t'\{y_1 \mapsto y\} \triangleq s'\{y_2 \mapsto y\}$. By the induction hypothesis, $X'(\vec{x}, y)\vartheta_2^n \preceq t'\{y_1 \mapsto y\}$ and $X'(\vec{x}, y)\vartheta_2^n \preceq s'\{y_1 \mapsto y\}$. Since $X(\vec{x})\vartheta_1^n = \lambda y.X'(\vec{x}, y)\vartheta_2^n$ and due to the way how y was chosen, we finally get $X(\vec{x})\vartheta_1^n \preceq \lambda y.t'\{y_1 \mapsto y\} = t$ and $X(\vec{x})\vartheta_1^n \preceq \lambda y.s'\{y_2 \mapsto y\} = s$.

Sol: We have $X(\vec{x})\vartheta_1^1 = Y(\vec{y})$ where Y is in the store. By the induction hypothesis, $Y(\vec{y})\vartheta_2^n \preceq t$ and $Y(\vec{y})\vartheta_2^n \preceq s$. Therefore, $X(\vec{x})\vartheta_1^n \preceq t$ and $X(\vec{x})\vartheta_1^n \preceq s$.

For Mer, there exists $Y(\vec{y}) : t' \triangleq s' \in S_0$ such that $\mathsf{match}(\{\vec{x}\}, \{\vec{y}\}, t \Rightarrow t', s \Rightarrow s')$ is a permuting matcher π, and $\vartheta_1^1 = \{Y \mapsto \lambda\vec{y}.X(\vec{x}\pi)\}$. By the induction hypothesis, $X(\vec{x})\vartheta_1^n = X(\vec{x})\vartheta_2^n \preceq t$ and $X(\vec{x})\vartheta_1^n = X(\vec{x})\vartheta_2^n \preceq s$. These imply that $X(\vec{x}\pi)\vartheta_1^n \preceq t'$ and $X(\vec{x}\pi)\vartheta_1^n \preceq s'$, which, together $Y\vartheta_1^n = X(\vec{x}\pi)$, yields $Y(\vec{y})\vartheta_1^n \preceq t'$and $Y(\vec{y})\vartheta_1^n \preceq s'$. ◂

Hence, the result computed by $\mathfrak{P}$ for $X : t \triangleq s$ generalizes both t and s. We call $X\sigma$, a *generalization of t and s computed by $\mathfrak{P}$*. Moreover, given a derivation $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S; \sigma$ in $\mathfrak{P}$, we say that

- σ is a *substitution computed by $\mathfrak{P}$ for $X : t \triangleq s$;*
- the restriction of σ on X, $\sigma|_X$, is an *anti-unifier of $X : t \triangleq s$ computed by $\mathfrak{P}$.*

▸ **Theorem 4.3** (Completeness). *Let $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$ be higher-order terms and $\lambda\vec{x}.s$ be a higher-order pattern such that $\lambda\vec{x}.s$ is a generalization of both $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$. Then $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$, where σ is an anti-unifier of $X : \lambda\vec{x}.t_1 \triangleq \lambda\vec{x}.t_2$ computed by $\mathfrak{P}$.*

Proof. By structural induction on s. We can assume without loss of generality that $\lambda\vec{x}.s$ is an lgg of $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$. We also assume that it is in the η-long β-normal form.

If s is a variable, then there are two cases: Either $s \in \vec{x}$, or $s \notin \vec{x}$. In the first case, we have $s = t_1 = t_2$. The Dec rule gives $\sigma = \{X \mapsto \lambda\vec{x}.s\}$ and, hence, $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma = s$. In the second case, either $\mathrm{Head}(t_1) \neq \mathrm{Head}(t_2)$, or $\mathrm{Head}(t_1) = \mathrm{Head}(t_2) \notin \vec{x}$. Sol is supposed to give us $\sigma = \{X \mapsto \lambda\vec{x}.X'(\vec{x'})\}$, where $\vec{x'}$ is a subsequence of $\vec{x}$ consisting of variables occurring freely in t_1 or in t_2. But $\vec{x'}$ should be empty, because otherwise s would not be just a variable (remember that $\lambda\vec{x}.s$ is an lgg of $\lambda\vec{x}.t_1$ and $\lambda\vec{x}.t_2$ in the η-long β-normal form). Hence, we have $\sigma = \{X \mapsto \lambda\vec{x}.X'\}$ and $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$, because $s\{s \mapsto X'\} = X(\vec{x})\sigma$.

If s is a constant c, then $t_1 = t_2 = c$. We can apply the Dec rule, obtaining $\sigma = \{X \mapsto \lambda\vec{x}.c\}$ and, hence, $s = c \preceq X(\vec{x})\sigma = c$. Therefore, $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$.

If $s = \lambda x.s'$, then t_1 and t_2 must have the forms $t_1 = \lambda x.t_1'$ and $t_2 = \lambda y.t_2'$, and s' must be an lgg of t_1' and t_2'. Abs gives a new system $\{X'(\vec{x}, x) : t_1' \triangleq t_2'\{x \mapsto y\}\}; \varnothing; \sigma_1$, where $\sigma_1 = \{X \mapsto \lambda\vec{x}, x.X'(\vec{x}, x)\}$. By the induction hypothesis, we can compute a substitution

σ_2 such that $\lambda\vec{x}, x.s' \preceq \lambda\vec{x}, x.X'(\vec{x}, x)\sigma_2$. Composing σ_1 and σ_2 into σ, we have $X(\vec{x})\sigma = \lambda x.X'(\vec{x}, x)\sigma_2$. Hence, we get $\lambda\vec{x}.s = \lambda\vec{x}.\lambda x.s' \preceq \lambda\vec{x}.\lambda x.X'(\vec{x}, x)\sigma_2 = \lambda\vec{x}.X(\vec{x})\sigma$.

Finally, assume that s is a compound term $h(s_1, \ldots, s_n)$. If $h \notin \vec{x}$ is a variable, then $s_1, \ldots, s_n$ are distinct variables from $\vec{x}$ (because $\lambda\vec{x}.s$ is a higher-order pattern). That means that $s_1, \ldots, s_n$ appear freely in t_1 or t_2. Moreover, either $\mathrm{Head}(t_1) \neq \mathrm{Head}(t_2)$, or $\mathrm{Head}(t_1) = \mathrm{Head}(t_2) = h$. In both cases, we can apply the Sol rule to obtain $\sigma = \{X \mapsto \lambda\vec{x}.Y(s_1, \ldots, s_n)\}$. Obviously, $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma = \lambda\vec{x}.Y(s_1, \ldots, s_n)$.

If $h \in \vec{x}$ or if it is a constant, then we should have $\mathrm{Head}(t_1) = \mathrm{Head}(t_2)$. Assume they have the forms $t_1 = h(t_1^1, \ldots, t_n^1)$ and $t_2 = h(t_1^2, \ldots, t_n^2)$. We proceed by the Dec rule, obtaining $\{Y_i(\vec{x}) : t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0$, where $\sigma_0 = \{X \mapsto \lambda\vec{x}.h(Y_1(\vec{x}), \ldots, Y_n(\vec{x}))\}$. By the induction hypothesis, we can construct derivations $\Delta_1, \ldots, \Delta_n$ computing the substitutions $\sigma_1, \ldots, \sigma_n$, respectively, such that $\lambda\vec{x}.s_i \preceq \lambda\vec{x}.Y_i(\vec{x})\sigma_i$ for $1 \leqslant i \leqslant n$. These derivations, together with the initial Dec step, can be combined into one derivation, of the form $\Delta = \{X(\vec{x}) : t_1 \triangleq t_2\}; \varnothing; \sigma_0 \Longrightarrow \{Y_i(\vec{x}) : t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0 \Longrightarrow^* \varnothing; S_n; \sigma_0\sigma_1 \cdots \sigma_n$.

Let for any term t, $t|_p$ denote the subterm of t at position p. If s does not contain duplicate variables free in $\lambda\vec{x}.s$, then the construction of Δ and the fact that $\lambda\vec{x}.s_i \preceq \lambda\vec{x}.Y_i(\vec{x})\sigma_i$ for $1 \leqslant i \leqslant n$ guarantee $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma_0\sigma_1 \cdots \sigma_n$. If s contains duplicate variables free in $\lambda\vec{x}.s$ (e.g., of the form $\lambda\vec{u_1}.Z(\vec{z_1})$ and $\lambda\vec{u_2}.Z(\vec{z_2})$, where $\vec{z_1}$ and $\vec{z_2}$ have the same length) at positions p_1 and p_2, it indicates that

(a) $t_1|_{p_1}$ and $t_1|_{p_2}$ differ from each other by a permutation of variables bound in t_1,
(b) $t_2|_{p_1}$ and $t_2|_{p_2}$ differ from each other by the same (modulo variable renaming) permutation of variables bound in t_2,
(c) the path to p_1 is the same (modulo bound variable renaming) in t_1 and t_2. It equals (modulo bound variable renaming) the path to p_1 in s, and
(d) the path to p_2 is the same (modulo bound variable renaming) in t_1 and t_2. It equals (modulo bound variable renaming) the path to p_2 in s.

Then, because of (c) and (d), we should have two AUPs in S_n: One, between (renamed variants of) $t_1|_{p_1}$ and $t_2|_{p_1}$, and the other one between (renamed variants of) $t_1|_{p_2}$ and $t_2|_{p_2}$. The possible renaming of variables is caused by the fact that Abs might have been applied to obtain the AUPs. Let those AUPs be $Z(\vec{z_1}) : r_1^1 \triangleq r_1^2$ and $Z'(\vec{z_2}) : r_2^1 \triangleq r_2^2$. The conditions (a) and (b) make sure that $\mathsf{match}(\{\vec{z_1}\}, \{\vec{z_2}\}, \{r_1^1 \Rightarrow r_2^1, r_1^2 \Rightarrow r_2^2\})$ is a permuting matcher π, which means that we can apply the rule Mer with the substitution $\sigma_1' = \{Z' \mapsto \lambda\vec{z_2}.Z(\vec{z_1}\pi)\}$. We can repeat this process for all duplicated variables in s, extending Δ to the derivation $\Delta' = \{X(\vec{x}) : t_1 \triangleq t_2\}; \varnothing; \sigma_0 \Longrightarrow \{Y_i(\vec{x}) : t_i^1 \triangleq t_i^2 \mid 1 \leqslant i \leqslant n\}; \varnothing; \sigma_0 \Longrightarrow^* \varnothing; S_n; \sigma_0\sigma_1 \cdots \sigma_n \Longrightarrow^* \varnothing; S_{n+m}; \sigma_0\sigma_1 \cdots \sigma_n\sigma_1' \cdots \sigma_m'$, where $\sigma_1', \ldots, \sigma_m'$ are substitutions introduced by the applications of the Mer rule. Let $\sigma = \sigma_0\sigma_1 \cdots \sigma_n\sigma_1' \cdots \sigma_m'$. By this construction, we have $\lambda\vec{x}.s \preceq \lambda\vec{x}.X(\vec{x})\sigma$, which finishes the proof. ◂

Depending which AUP is selected to perform a step, there can be different derivations in $\mathfrak{P}$ starting from the same AUP, leading to different generalizations. The next theorem states that all those generalizations are equivalent.

▸ **Theorem 4.4** (Uniqueness Modulo $\simeq$). *Let* $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S_1; \sigma_1$ *and* $\{X : t \triangleq s\}; \varnothing; \varnothing \Longrightarrow^* \varnothing; S_2; \sigma_2$ *be two maximal derivations in* $\mathfrak{P}$ *from* $X : t \triangleq s$. *Then* $X\sigma_1 \simeq X\sigma_2$.

Proof. It is not hard to notice that if it is possible to change the order of applications of rules (but sticking to the same selected AUPs for each rule) then the result remains the same: If $\Delta_1 = A_1; S_1; \sigma_1 \Longrightarrow_{\mathsf{R1}} A_2; S_2; \sigma_1\vartheta_1 \Longrightarrow_{\mathsf{R2}} A_3; S_3; \sigma_1\vartheta_1\vartheta_2$ and $\Delta_2 = A_1; S_1; \sigma_1 \Longrightarrow_{\mathsf{R2}}$

$A'_2; S'_2; \sigma_1\vartheta_2 \Longrightarrow_{\mathsf{R1}} A'_3; S'_3; \sigma_1\vartheta_2\vartheta_1$ are two two-step derivations, where R1 and R2 are (not necessarily different) rules and each of them transforms the same AUP(s) in both Δ_1 and Δ_2, then $A_3 = A'_3$, $S_3 = S'_3$, and $\sigma_1\vartheta_1\vartheta_2 = \sigma_1\vartheta_2\vartheta_1$ (modulo the names of fresh variables).

Decomposition, Abstraction, and Solve rules transform the selected AUP in a unique way. We show that it is irrelevant in which order we perform matching in the Merge rule.

Let $A; \{Z(\vec{z}) : t_1 \triangleq s_1, Y(\vec{y}) : t_2 \triangleq s_2\} \cup S; \sigma \Longrightarrow A; \{Z(\vec{z}) : t_1 \triangleq s_1\} \cup S; \sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\}$ be the merging step with $\pi = \mathsf{match}(\{\vec{z}\}, \{\vec{y}\}, \{t_1 \Rightarrow t_2, s_1 \Rightarrow s_2\})$. If we do it in the other way around, we would get the step $A; \{Z(\vec{z}) : t_1 \triangleq s_1, Y(\vec{y}) : t_2 \triangleq s_2\} \cup S; \sigma \Longrightarrow A; \{Y(\vec{y}) : t_2 \triangleq s_2\} \cup S; \sigma\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\mu)\}$, where $\mu = \mathsf{match}(\{\vec{y}\}, \{\vec{z}\}, \{t_2 \Rightarrow t_1, s_2 \Rightarrow s_1\})$. But $\mu = \pi^{-1}$, because of bijection.

Let $\vartheta_1 = \sigma\rho_1$ with $\rho_1 = \{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\}$ and $\vartheta_2 = \sigma\rho_2$ with $\rho_2 = \{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\}$. Our goal is to prove that $X\vartheta_1 \simeq X\vartheta_2$. For this, we have to prove two inequalities: $X\vartheta_1 \preceq X\vartheta_2$ and $X\vartheta_2 \preceq X\vartheta_1$. To show $X\vartheta_1 \preceq X\vartheta_2$, we first need to prove the equality:

$$\lambda\vec{y}.Z(\vec{z}\pi)\rho_2 = \lambda\vec{y}.Y(\vec{y}). \tag{1}$$

Its left hand side is transformed as $\lambda\vec{y}.Z(\vec{z}\pi)\rho_2 = \lambda\vec{y}.Z(\vec{z}\pi)\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = \lambda\vec{y}.(\lambda\vec{z}.Y(\vec{y}\pi^{-1})(\vec{z}\pi))$. β-reduction of $\lambda\vec{z}.Y(\vec{y}\pi^{-1})(\vec{z}\pi)$ replaces each occurrence of $z_i \in \vec{z}$ in $Y(\vec{y}\pi^{-1})$ with $z_i\pi$, which is the same as applying π to $Y(\vec{y}\pi^{-1})$. Since $\vec{y}\pi^{-1}\pi = \vec{y}$, we get $\lambda\vec{y}.(\lambda\vec{z}.Y(\vec{y}\pi^{-1})(\vec{z}\pi)) = \lambda\vec{y}.Y(\vec{y}\pi^{-1}\pi) = \lambda\vec{y}.Y(\vec{y})$ and (1) is proved.

Next, starting from $X\vartheta_1\rho_2$, we can transform it as $X\vartheta_1\rho_2 = X\sigma\rho_1\rho_2 = X\sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\rho_2, Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} =_{\text{by (1)}} X\sigma\{Y \mapsto \lambda\vec{y}.Z(\vec{z}\pi)\rho_2, Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y}), Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\}$. At this step, since the equality $=$ is $\alpha\beta\eta$-equivalence, we can omit the application of the substitution $\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}$ and proceed: $X\sigma\{Y \mapsto \lambda\vec{y}.Y(\vec{y})\}\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\{Z \mapsto \lambda\vec{z}.Y(\vec{y}\pi^{-1})\} = X\sigma\rho_2X\vartheta_2$. Hence, we got $X\vartheta_1\rho_2 = X\vartheta_2$, which implies $X\vartheta_1 \preceq X\vartheta_2$.

$X\vartheta_2 \preceq X\vartheta_1$ can be proved analogously. Hence, $X\vartheta_1 \simeq X\vartheta_2$, which means that it is irrelevant in which order we perform matching in the Merge rule. Therefore, no matter how different derivations are constructed, the computed generalizations are equivalent. ◂

Hence, for given terms t and s, the anti-unification algorithm $\mathfrak{P}$ computes their generalization, a higher-order pattern, which is less general than any other higher-order pattern which generalizes t and s. The next theorem is about its complexity:

▸ **Theorem 4.5** (Complexity of $\mathfrak{P}$). *The algorithm $\mathfrak{P}$, when using $\mathfrak{M}$ to compute permuting matchers, has space complexity $O(n)$ and time complexity $O(n^3)$, where n is the size (the number of symbols) of input.*

Proof. We can keep the substitutions in the systems in triangular form. Then the size of systems is linear in the size of input. Only at the end we will apply the computed anti-unifier to the corresponding generalization variable to return the generalization: Having the substitution $[X \mapsto t_0, Y_1 \mapsto t_1, \ldots, Y_n \mapsto t_n]$, we need to compute $t_0\{Y_1 \mapsto t_1\} \cdots \{Y_n \mapsto t_n\}$. Its size does not exceed the size on the input. Hence, the space complexity is linear.

For proving the cubic time complexity, we can assume that the applications of the Mer rule are postponed till the end. The number of application of the other rules is bounded by the size of the input. Abs involves renaming which can be done in linear time. Sol requires selection of variables that occur freely in terms, which also needs linear time. Composition of substitutions is just appending a new binding at the end of the existing triangular substitution. As for the Mer rule, it can be called at most quadratic number of times. At each application it calls $\mathfrak{M}$ which itself requires linear time. Hence, the cubic complexity of applications of Mer dominates the complexity of applications of the other rules. The

last step, constructing the generalization $t_0\{Y_1 \mapsto t_1\}\cdots\{Y_n \mapsto t_n\}$ from the computed triangular substitution, requires linear number of substitution applications. Each application traverses the term, replaces all occurrences of Y_i with t_i, and performs β-reduction (i.e, bound variable permutation). Traversal, replacement, and β-reduction can take at most quadratic time. Therefore, the complexity of this last step is also cubic. It implies that $\mathfrak{P}$ has the $O(n^3)$ time complexity.

Note that if the input does not satisfy the condition each bound variable to be unique (on which both $\mathfrak{P}$ and $\mathfrak{M}$ rely), we can rename the variables before calling $\mathfrak{P}$. It can be done in linear time, using a "chained-like" hash table whose buckets are stacks (instead of linked lists of chained hash tables) for variable renaming, and traversing the terms in preorder. ◂

5 Final Remarks

One can observe that $\mathfrak{P}$ can be adapted with a relatively little effort to work on untyped terms (cf. the formulation of the unification algorithm both for untyped and simply-typed patterns in [27]). One thing to be added is lazy η-expansion: The AUP of the form $X(\vec{x}) : \lambda y.t \triangleq h(s_1, \ldots, s_m)$ should be transformed into $X(\vec{x}) : \lambda y.t \triangleq \lambda z.h(s_1, \ldots, s_m, z)$ for a fresh z. (Dually for abstractions in the right hand side.) The expansion should be performed both in $\mathfrak{P}$ and $\mathfrak{M}$. In addition, Sol needs an extra condition for the case when $\mathrm{Head}(t) = \mathrm{Head}(s)$ but the terms have different number of arguments such as, e.g., in $f(a, x)$ and $f(b, x, y)$.

The anti-unification algorithm has been implemented (both for simply-typed and untyped terms, without perfect hashing) in Java. It can be used online or can be downloaded freely from `http://www.risc.jku.at/projects/stout/software/hoau.php`.

As for the related topics, we would mention nominal anti-unification. Several authors explored relationship between nominal terms and higher-order patterns (see, e.g., [11, 13, 20, 21] among others), proposing translations between them in the context of unification. However, it is not immediately clear how to reuse those translations for anti-unification, in particular, how to get nominal generalizations from pattern generalizations.

Studying anti-unification in the calculi with more complex type systems, such as the extension of the system F with subtyping $F_{<:}$ [10], would be a very interesting direction of future work, because it may have applications in clone detection and refactoring for the functional programming languages in the ML family.

References

1 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In *LOPSTR*, volume 5438 of *LNCS*, pages 24–39. Springer, 2008.

2 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. Order-sorted generalization. *Electr. Notes Theor. Comput. Sci.*, 246:27–38, 2009.

3 E. Armengol and E. Plaza. Bottom-up induction of feature terms. *Machine Learning*, 41(3):259–294, 2000.

4 H. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

5 A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. A variant of higher-order anti-unification. Technical Report 12-19, RISC, Johannes Kepler University Linz, 2012. `http://www.risc.jku.at/publications/download/risc_4675/hoau.pdf`.

6 P. Bulychev. Duplicate code detection using Clone Digger. *Python Mag.*, 9:18–24, 2008.

7 P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*, 2009.

8 P. E. Bulychev, E. V. Kostylev, and V. A. Zakharov. Anti-unification algorithms and their applications in program analysis. In *Ershov Memorial Conference*, volume 5947 of *LNCS*, pages 413–423. Springer, 2009.

9 J. Burghardt. E-generalization using grammars. *Artif. Intell.*, 165(1):1–35, 2005.

10 L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994.

11 J. Cheney. Relating higher-order pattern unification and nominal unification. In *UNIF'05*, pages 104–119, 2005.

12 G. Dowek. Higher-order unification and matching. In *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier and MIT Press, 2001.

13 G. Dowek, M. J. Gabbay, and D. P. Mulligan. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Logic Journal of the IGPL*, 18(6):769–822, 2010.

14 C. Feng and S. Muggleton. Towards inductive generalization in higher order logic. In *ML*, pages 154–162. Morgan Kaufmann, 1992.

15 R. W. Hasker. *The Replay of Program Derivations*. PhD thesis, University of Illionois at Urbana-Champaign, 1995.

16 K. Hirata, T. Ogawa, and M. Harao. Generalization algorithms for second-order terms. In *ILP*, volume 3194 of *LNCS*, pages 147–163. Springer, 2004.

17 G. Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

18 U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In *AUS-AI*, volume 4830 of *LNCS*, pages 273–282. Springer, 2007.

19 T. Kutsia, J. Levy, and M. Villaret. Anti-unification for unranked terms and hedges. In *RTA*, volume 10 of *LIPIcs*, pages 219–234, 2011.

20 J. Levy and M. Villaret. Nominal unification from a higher-order perspective. In *RTA*, volume 5117 of *LNCS*, pages 246–260. Springer, 2008.

21 J. Levy and M. Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012.

22 H. Li and S. J. Thompson. Similar code detection and elimination for Erlang programs. In *PADL*, volume 5937 of *LNCS*, pages 104–118. Springer, 2010.

23 J. Lu, J. Mylopoulos, M. Harao, and M. Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.

24 G. Mendel-Gleason. *Types and Verification for Infinite State Systems*. PhD thesis, Dublin City University, 2012.

25 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Log. Comput.*, 1(4):497–536, 1991.

26 G. Nadathur and N. Linnell. Practical higher-order pattern unification with on-the-fly raising. In *ICLP*, volume 3668 of *LNCS*, pages 371–386. Springer, 2005.

27 T. Nipkow. Functional unification of higher-order patterns. In *LICS*, pages 64–74. IEEE Computer Society, 1993.

28 F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

29 G. D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.

30 J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.

31 U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *LNCS*. Springer, 2003.

Over-approximating Descendants by Synchronized Tree Languages

Yohan Boichut, Jacques Chabin, and Pierre Réty

LIFO – Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
{yohan.boichut, jacques.chabin, pierre.rety}@univ-orleans.fr

Abstract

Over-approximating the descendants (successors) of a initial set of terms by a rewrite system is used in verification. The success of such verification methods depends on the quality of the approximation. To get better approximations, we are going to use non-regular languages. We present a procedure that always terminates and that computes an over-approximation of descendants, using synchronized tree-(tuple) languages expressed by logic programs.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases rewriting systems, non-regular approximations, logic programming, tree languages, descendants

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.128

1 Introduction

Given an initial set of terms I, computing the descendants (successors) of I by a rewrite system R is used in the verification domain, for example to check cryptographic protocols or Java programs [2, 8, 10, 9]. Let $R^*(I)$ denote the set of descendants of I, and consider a set Bad of *undesirable* terms. Thus, if a term of Bad is reached from I, i.e. $R^*(I) \cap Bad \neq \emptyset$, it means that the protocol or the program is flawed. In general, it is not possible to compute $R^*(I)$ exactly. Instead, we compute an over-approximation App of $R^*(I)$ (i.e. $App \supseteq R^*(I)$), and check that $App \cap Bad = \emptyset$, which ensures that the protocol or the program is correct.

Most often, I, App and Bad have been considered as regular tree languages, recognized by finite tree automata. In the general case, $R^*(I)$ is not regular, even if I is. Moreover, the expressiveness of regular languages is poor, and the over-approximation App may not be precise enough, and we may have $App \cap Bad \neq \emptyset$ whereas $R^*(I) \cap Bad = \emptyset$. In other words, the protocol is correct, but we cannot prove it. Some work has proposed CEGAR-techniques (Counter-Example Guided Approximation Refinement) in order to conclude as often as possible [2, 4, 6]. However, in some cases, no regular over-approximation works, whatever the quality of the approximation is [5].

To overcome this theoretical limit, we want to use more expressive languages to express the over-approximation, i.e. non-regular ones. However, to be able to check that $App \cap Bad = \emptyset$, we need a class of languages closed under intersection and whose emptiness is decidable. Actually, since we still assume that Bad is regular, closure under intersection with a regular language is enough. The class of context-free tree languages has these properties, and an over-approximation of descendants using context-free tree languages has been proposed in [14]. This class of languages is quite interesting, however it cannot express relations (or countings) in terms between independent branches, except if there are only unary symbols and constants. For example, let $R = \{f(x) \to c(x, x)\}$ and the infinite set $I = \{f(t)\}$ where t denotes any term composed with the binary symbol g and constant b. Then $R^*(I) = I \cup \{c(t, t)\}$, which is not a context-free language [1, 13].

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 128–142

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We want to use another class of languages that has the needed properties, and that can express relations between independent branches: the synchronized tree-(tuple) languages [15, 12], which were finally expressed thanks to logic programs (Horn clauses) [16, 17]. This class has the same properties as context-free tree languages: closure under union, closure under intersection with a regular language (in quadratic time), decidability of membership and emptiness (in linear time). Both include regular languages, however they are different. The example given above is not context-free, but synchronized. The language $\{s^n(p^n(a))\}$ (where s^n means that s occurs n times vertically) is context-free, but it is not synchronized. $\{c(s^n(a), p^n(a))\}$ belongs to both classes (note that s and p are unary).

In this paper, we propose a procedure that always terminates and that computes an over-approximation of the descendants obtained by a left-linear rewrite system, using synchronized tree-(tuple) languages expressed by logic programs. Note that the left-linearity of rewrite systems (or transducers) is a usual restriction, see [2, 6, 8, 10, 9]. Nevertheless, such rewrite systems are still Turing complete [7].

The paper is organized as follows: classical notations and notions manipulated throughout the paper are introduced in Section 2. Our main contribution, i.e. computing approximations using synchronized languages, is explained in Section 3. Finally, in Section 4 our technique is applied on two pertinent examples: an example illustrating a non-regular approximation of a non-regular set of terms, and another one that cannot be handled by any regular approximation.

2 Preliminaries

Consider a *finite ranked alphabet* Σ and a set of variables Var. Each symbol $f \in \Sigma$ has a unique arity, denoted by $ar(f)$. The notions of *first-order term, position, substitution,* are defined as usual. Given σ and σ' two substitutions, $\sigma \circ \sigma'$ denotes the substitution such that for any variable x, $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. T_Σ denotes the set of ground terms (without variables) over Σ. For a term t, $Var(t)$ is the set of variables of t, $Pos(t)$ is the set of positions of t. For $p \in Pos(t)$, $t(p)$ is the symbol of $\Sigma \cup Var$ occurring at position p in t, and $t|_p$ is the subterm of t at position p. The term $t[t']_p$ is obtained from t by replacing the subterm at position p by t'. $PosVar(t) = \{p \in Pos(t) \mid t(p) \in Var\}$, $PosNonVar(t) = \{p \in Pos(t) \mid t(p) \notin Var\}$. Note that if $p \in PosNonVar(t)$, $t|_p = f(t_1, \ldots, t_n)$, and $i \in \{1, \ldots, n\}$, then $p.i$ is the position of t_i in t. For $p, p' \in Pos(t)$, $p < p'$ means that p occurs in t strictly above p'. Let t, t' be terms, t is *more general than* t' (denoted $t \leq t'$) if there exists a substitution ρ s.t. $\rho(t) = t'$. Let σ, σ' be substitutions, σ is *more general than* σ' (denoted $\sigma \leq \sigma'$) if there exists a substitution ρ s.t. $\rho \circ \sigma = \sigma'$.

A *rewrite rule* is an oriented pair of terms, written $l \to r$. We always assume that l is not a variable, and $Var(r) \subseteq Var(l)$. A *rewrite system* R is a finite set of rewrite rules. *lhs* stands for left-hand-side, *rhs* for right-hand-side. The rewrite relation $\to_R$ is defined as follows: $t \to_R t'$ if there exist a position $p \in PosNonVar(t)$, a rule $l \to r \in R$, and a substitution θ s.t. $t|_p = \theta(l)$ and $t' = t[\theta(r)]_p$. $\to_R^*$ denotes the reflexive-transitive closure of $\to_R$. t' is a *descendant* of t if $t \to_R^* t'$. If E is a set of ground terms, $R^*(E)$ denotes the set of descendants of elements of E.

In the following, we consider the framework of *pure logic programming*, and the class of synchronized tree-tuple languages defined by CS-clauses [16, 17]. Given a set $Pred$ of *predicate* symbols; *atoms*, *goals*, *bodies* and *Horn-clauses* are defined as usual. Note that both *goals* and *bodies* are sequences of atoms. We will use letters G or B for sequences of atoms, and A for atoms. Given a goal $G = A_1, \ldots, A_k$ and positive integers i, j, we define

$G|_i = A_i$ and $G|_{i.j} = (A_i)|_j = t_j$ where $A_i = P(t_1, \ldots, t_n)$.

▶ **Definition 1.** Let B be a sequence of atoms. B is *flat* if for each atom $P(t_1, \ldots, t_n)$ of B, all terms $t_1, \ldots, t_n$ are variables. B is *linear* if each variable occurring in B (possibly at sub-term position) occurs only once in B. Note that the empty sequence of atoms (denoted by $\emptyset$) is flat and linear.
A *CS-clause*[1] is a Horn-clause $H \leftarrow B$ s.t. B is flat and linear. A *CS-program* $Prog$ is a logic program composed of CS-clauses.

Given a predicate symbol P of arity n, the tree-(tuple) language generated by P is $L(P) = \{\vec{t} \in (T_\Sigma)^n \mid P(\vec{t}) \in Mod(Prog)\}$, where T_Σ is the set of ground terms over the signature Σ and $Mod(Prog)$ is the least Herbrand model of $Prog$. $L(P)$ is called *Synchronized language*.

The following definition describes the different kinds of CS-clauses that can occur.

▶ **Definition 2.** A CS-clause $P(t_1, \ldots, t_n) \leftarrow B$ is :
- *empty* if $\forall i \in \{1, \ldots, n\}$, t_i is a variable.
- *normalized* if $\forall i \in \{1, \ldots, n\}$, t_i is a variable or contains only one occurrence of function-symbol. A CS-program is *normalized* if all its clauses are normalized.
- *preserving* if $Var(P(t_1, \ldots, t_n)) \subseteq Var(B)$. A CS-program is *preserving* if all its clauses are preserving.
- *synchronizing* if B is composed of only one atom.

▶ **Example 3.** The CS-clause $P(x, y, z) \leftarrow G(x, y, z)$ is empty, normalized, and preserving (x, y, z are variables). The CS-clause $P(f(x), y, g(x, z)) \leftarrow G(x, y)$ is normalized and non-preserving. Both clauses are synchronizing.

Given a CS-program, we focus on two kinds of derivations: a classical one based on unification and a rewriting one based on matching and a rewriting process.

▶ **Definition 4.** Given a logic program $Prog$ and a sequence of atoms G,
- G derives into G' by a *resolution* step if there exist a clause[2] $H \leftarrow B$ in $Prog$ and an atom $A \in G$ such that A and H are unifiable by the most general unifier σ (then $\sigma(A) = \sigma(H)$) and $G' = \sigma(G)[\sigma(A) \leftarrow \sigma(B)]$. It is written $G \rightsquigarrow_\sigma G'$.
- G *rewrites* into G' if there exist a clause $H \leftarrow B$ in $Prog$, an atom $A \in G$, and a substitution σ, such that $A = \sigma(H)$ (A is not instantiated by σ) and $G' = G[A \leftarrow \sigma(B)]$. It is written $G \rightarrow_\sigma G'$.

▶ **Example 5.** Let $Prog = \{P(x_1, g(x_2)) \leftarrow P'(x_1, x_2).\ P(f(x_1), x_2) \leftarrow P''(x_1, x_2).\}$, and consider $G = P(f(x), y)$. Thus, $P(f(x), y)) \rightsquigarrow_{\sigma_1} P'(f(x), x_2)$ with $\sigma_1 = [x_1/f(x), y/g(x_2)]$ and $P(f(x), y)) \rightarrow_{\sigma_2} P''(x, y)$ with $\sigma_2 = [x_1/x, x_2/y]$.

We consider the transitive closure $\rightsquigarrow^+$ and the reflexive-transitive closure $\rightsquigarrow^*$ of $\rightsquigarrow$.

For both derivations, given a logic program $Prog$ and three sequences of atoms G_1, G_2 and G_3 :
- if $G_1 \rightsquigarrow_{\sigma_1} G_2$ and $G_2 \rightsquigarrow_{\sigma_2} G_3$ then one has $G_1 \rightsquigarrow^*_{\sigma_2 \circ \sigma_1} G_3$;
- if $G_1 \rightarrow_{\sigma_1} G_2$ and $G_2 \rightarrow_{\sigma_2} G_3$ then one has $G_1 \rightarrow^*_{\sigma_2 \circ \sigma_1} G_3$.

[1] In former papers, synchronized tree-tuple languages were defined thanks to sorts of grammars, called constraint systems. Thus "CS" stands for Constraint System.
[2] We assume that the clause and G have distinct variables.

In the remainder of the paper, given a set of CS-clauses $Prog$ and two sequences of atoms G_1 and G_2, $G_1 \rightsquigarrow^*_{Prog} G_2$ (resp. $G_1 \rightarrow^*_{Prog} G_2$) also denotes that G_2 can be derived (resp. rewritten) from G_1 using clauses of $Prog$.

It is well known that resolution is complete.

▶ **Theorem 6.** *Let A be a ground atom. $A \in Mod(Prog)$ iff $A \rightsquigarrow^*_{Prog} \emptyset$.*

▶ **Example 7.** Let $A = P(f(g(a)), g(a), c)$ and $A' = P'(f(g(a)), h(c))$ be two ground atoms. Let $Prog$ be the CS-program defined by:
$Prog = \{P(f(g(x)), y, c) \leftarrow P_1(x), P_2(y).\ P_1(a) \leftarrow .\ \ P_2(g(x)) \leftarrow P_1(x).\ P'(f(x), u(z)) \leftarrow .\}$
Thus, $A \in Mod(Prog)$ and $A' \notin Mod(Prog)$.

Note that for any atom A, if $A \rightarrow B$ then $A \rightsquigarrow B$. If in addition $Prog$ is preserving, then $Var(A) \subseteq Var(B)$. On the other hand, $A \rightsquigarrow_\sigma B$ implies $\sigma(A) \rightarrow B$. Consequently, if A is ground, $A \rightsquigarrow B$ implies $A \rightarrow B$.

The following lemma focuses on a preserving property of the relation $\rightsquigarrow$.

▶ **Lemma 8.** *Let $Prog$ be a CS-program, and G be a sequence of atoms. Let $|G|_\Sigma$ denote the number of occurrences of function-symbols in G. If G is linear and $G \rightsquigarrow^* G'$, then G' is also linear and $|G'|_\Sigma \leq |G|_\Sigma$.*
Consequently, if G is flat and linear, then G' is also flat and linear.

Proof. Let $G = A^1 \dots A^k$ be a linear sequence of atoms and suppose that $G \rightsquigarrow_\sigma G'$.
Then there exist an atom $A^i(s_1, \dots, s_n)$ of G and a CS-clause $A^i(t_1, \dots, t_n) \leftarrow B$ in $Prog$ such that $G' = \sigma(G)[\sigma(A^i) \leftarrow \sigma(B)]$. As G is linear and σ is the most general unifier between $A^i(s_1, \dots, s_n)$ and $A^i(t_1, \dots, t_n)$, σ does not instantiate variables from $A^1, \dots, A^{i-1}, A^{i+1}, \dots A^k$. So $G' = A^1, \dots, A^{i-1}, \sigma(B), A^{i+1}, \dots A^k$.

G' is not linear only if $\sigma(B)$ is not linear. As B is linear, $\sigma(B)$ is not linear would require that two distinct variables x_{j_1}, x_{j_2} from B are instantiated by two terms containing a same variable $y \in Var(\sigma(x_{j_1}) \cap Var(\sigma(x_{j_1}))$. Since σ is the most general unifier, x_{j_1}, x_{j_2} are also in $Var(A^i(t_1, \dots, t_n))$ (σ does not instantiate extra variables). Then y occurs at least twice in $A^i(s_1, \dots, s_n)$ (the atom of goal G), which is impossible since G is linear. Consequently G' is linear.

By contradiction: to obtain $|G'|_\Sigma > |G|_\Sigma$, we must have in $\sigma(B)$ a duplication of a non-variable subterm of $\sigma((A^i(s_1, \dots, s_n))$ (because B is flat), which is not possible because B and $A^i(s_1, \dots, s_n)$ are linear and σ is the most general unifier.

The result trivially extends to the case of several steps $G \rightsquigarrow^* G'$. ◀

▶ **Example 9.** Let $Prog = \{P(g(x), f(x)) \leftarrow P_1(x)\}$ and $G = P(g(f(y)), z)$. Then $G \rightsquigarrow G'$ with $G' = P_1(f(y))$, and G' is linear. Moreover, $|G'|_\Sigma \leq |G'|_\Sigma$ with $\Sigma = \{f^{\setminus 1}, a^{\setminus 0}\}$.

3 Computing Descendants

Given a CS-program $Prog$ and a left-linear rewrite system R, we propose a technique allowing us to compute a CS-program $Prog'$ such that $R^*(Mod(Prog)) \subseteq Mod(Prog')$. First of all, a notion of critical pairs is introduced in Section 3.1. Roughly speaking, this notion makes the detection of uncovered rewriting steps possible. Critical pair detection is at the heart of the technique. Thus, in Section 3.2 some restrictions are underlined on CS-programs in order to make the number of critical pairs finite. Moreover, when a CS-program does not fit these restrictions, we have proposed a technique in order to transform such a CS-program into another one of the expected form (REMOVE CYCLES in Fig.1). The detected critical

pairs lead to a set of CS-clauses to be added in the current CS-program. However, they may not be in the expected form i.e. normalized CS-clauses. Indeed, one of the restritions set in Section 3.2 is that the CS-program has to be normalized. So, we propose in Section 3.3 an algorithm providing normalized CS-clauses from non-normalized ones. Finally, in Section 3.4, our main contribution, i.e. the computation of an over-approximating CS-program, is fully described.

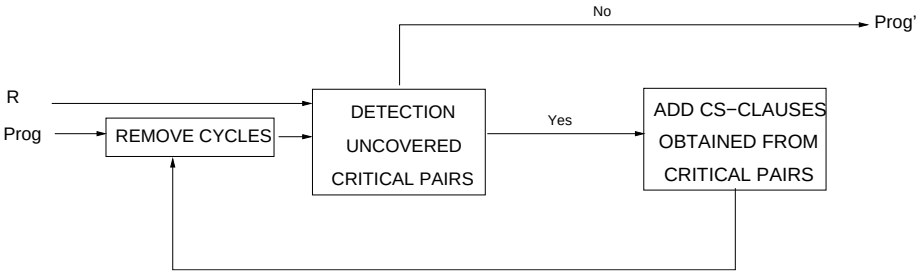


Figure 1 An overview of our contribution.

3.1 Critical pairs

The notion of critical pair is at the heart of our technique. Indeed, it allows us to add CS-clauses into the current CS-program in order to cover rewriting steps. This notion is described in Definition 10.

▶ **Definition 10.** Let $Prog$ be a CS-program and $l \to r$ be a left-linear rewrite rule. Let $x_1, \ldots, x_n$ be distinct variables s.t. $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$. If there are P and k s.t. $P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \rightsquigarrow_\theta^+ G$ where resolution is applied only on non-flat atoms, G is flat, and the clause $P(t_1, \ldots, t_n) \leftarrow B$ used during the first step of this derivation satisfies t_k is not a variable[3], then the clause $\theta(P(x_1, \ldots, x_{k-1}, r, x_{k+1}, \ldots, x_n)) \leftarrow G$ is called *critical pair*.

▶ Remark. Since l is linear, $P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n)$ is linear, and thanks to Lemma 8 G is linear, then a critical pair is a CS-clause. Moreover, if $Prog$ is preserving then a critical pair is a preserving CS-clause[4].

▶ **Example 11.** Let $Prog$ be the normalized and preserving CS-program defined by:

$$Prog = \{P(c(x), c(x), y) \leftarrow Q(x, y). \quad Q(a, b) \leftarrow . \quad Q(c(x), y) \leftarrow Q(x, y)\}.$$

and consider the left-linear rewrite rule: $c(c(x')) \to h(h(x'))$. Recall that for all goals G, G', the step $G \to G'$ means that $G \rightsquigarrow_\sigma G'$ where σ does not instantiate the variables of G. Thus $P(c(c(x')), y', z') \rightsquigarrow_\theta Q(c(x'), y) \to Q(x', y)$ where $\theta = [x/c(x'), y'/c(c(x')), z'/y]$. It generates the critical pair $P(h(h(x')), c(c(x')), y) \leftarrow Q(x', y)$. There are also two other critical pairs: $P(c(c(x')), h(h(x')), y) \leftarrow Q(x', y)$ and $Q(h(h(x')), y) \leftarrow Q(x', y)$.

---

[3] In other words, the overlap of l on the clause head $P(t_1, \ldots, t_n)$ is done at a non-variable position.

[4] We have $\theta(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n)) \to^* G$, and since $Prog$ is preserving $Var(\theta(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n))) \subseteq Var(G)$. Since $Var(r) \subseteq Var(l)$ we have $Var(\theta(P(x_1, \ldots, x_{k-1}, r, x_{k+1}, \ldots, x_n))) \subseteq Var(G)$.

However, some of the detected critical pairs are not so *critical* since they are already covered by the current CS-program. These critical pairs are said to be convergent.

▶ **Definition 12.** A critical pair $H \leftarrow B$ is said *convergent* if $H \rightarrow^*_{Prog} B$.

▶ **Example 13.** The three critical pairs detected in Example 11 are not convergent in $Prog$.

So, here we come to Theorem 14, i.e. the corner stone making our approach sound. Indeed, given a rewrite system R and CS-program $Prog$, if every critical pair that can be detected is convergent, then for any set of terms I such that $I \subseteq Mod(Prog)$, $Mod(Prog)$ is an over-approximation of the set of terms reachable by R from I.

▶ **Theorem 14.** *Let $Prog$ be a normalized and preserving CS-program and R be a left-linear rewrite system.*
If all critical pairs are convergent, then $Mod(Prog)$ is closed under rewriting by R, i.e. $(A \in Mod(Prog) \wedge A \rightarrow^*_R A') \implies A' \in Mod(Prog)$.

Proof. Let $A \in Mod(Prog)$ s.t. $A \rightarrow_{l \rightarrow r} A'$. Then $A|_i = C[\sigma(l)]$ for some $i \in \mathbb{N}$ and $A' = A[i \leftarrow C[\sigma(r)]$.
Since resolution is complete, $A \rightsquigarrow^* \emptyset$. Since $Prog$ is normalized and preserving, resolution consumes symbols in C one by one, thus $G_0 = A \rightsquigarrow^* G_k \rightsquigarrow^* \emptyset$ and there exists an atom $A'' = P(t_1, \ldots, t_n)$ in G_k and j s.t. $t_j = \sigma(l)$ and the top symbol of t_j is consumed during the step $G_k \rightsquigarrow G_{k+1}$. Consider new variables $x_1, \ldots, x_n$ s.t. $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$, and let us define the substitution σ' by $\forall i$, $\sigma'(x_i) = t_i$ and $\forall x \in Var(l)$, $\sigma'(x) = \sigma(x)$. Then $\sigma'(P(x_1, \ldots, x_{j-1}, l, x_{j+1}, \ldots, x_n)) = A''$, and according to resolution (or narrowing) properties $P(x_1, \ldots, l, \ldots, x_n) \rightsquigarrow^*_\theta \emptyset$ and $\theta \leq \sigma'$.
This derivation can be decomposed into : $P(x_1, \ldots, l, \ldots, x_n) \rightsquigarrow^*_{\theta_1} G' \rightsquigarrow_{\theta_2} G \rightsquigarrow^*_{\theta_3} \emptyset$ where $\theta = \theta_3 \circ \theta_2 \circ \theta_1$, and s.t. G' is not flat and G is flat[5]. $P(x_1, \ldots, l, \ldots, x_n) \rightsquigarrow^*_{\theta_1} G' \rightsquigarrow_{\theta_2} G$ can be commuted into $P(x_1, \ldots, l, \ldots, x_n) \rightsquigarrow^*_{\gamma_1} B' \rightsquigarrow_{\gamma_2} B \rightsquigarrow^*_{\gamma_3} G$ s.t. B is flat, B' is not flat, and within $P(x_1, \ldots, l, \ldots, x_n) \rightsquigarrow^*_{\gamma_1} B' \rightsquigarrow_{\gamma_2} B$ resolution is applied only on non-flat atoms, and we have $\gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_2 \circ \theta_1$. Then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \leftarrow B$ is a critical pair. By hypothesis, it is convergent, then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* B$. Note that $\gamma_3(B) \rightarrow^* G$ and recall that $\theta_3 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_3 \circ \theta_2 \circ \theta_1 = \theta$. Then $\theta(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \theta_3(G) \rightarrow^* \emptyset$, and since $\theta \leq \sigma'$ we get $P(t_1, \ldots, \sigma(r), \ldots, t_n) = \sigma'(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \emptyset$. Therefore $A' \rightsquigarrow^* G_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)] \rightsquigarrow^* \emptyset$, hence $A' \in Mod(Prog)$.
By trivial induction, the proof can be extended to the case of several rewrite steps. ◀

If $Prog$ is not normalized, Theorem 14 does not hold.

▶ **Example 15.** Let $Prog = \{P(c(f(a))) \leftarrow\}$ and $R = \{f(a) \rightarrow b\}$. All critical pairs are convergent since there is no critical pair. $P(c(f(a))) \in Mod(Prog)$ and $P(c(f(a))) \rightarrow_R P(c(b))$. However there is no resolution step issued from $P(c(b))$, then $P(c(b)) \notin Mod(Prog)$.

If $Prog$ is not preserving, Theorem 14 does not hold.

▶ **Example 16.** Let $Prog = \{P(c(x), c(x), y) \leftarrow Q(y).\ \ Q(a) \leftarrow\}$, and $R = \{f(b) \rightarrow b\}$. All critical pairs are convergent since there is no critical pair.
$P(c(f(b)), c(f(b)), a) \rightarrow_{Prog} Q(a) \rightarrow_{Prog} \emptyset$, then $P(c(f(b)), c(f(b)), a) \in Mod(Prog)$. On the other hand, $P(c(f(b)), c(f(b)), a) \rightarrow_R P(c(b), c(f(b)), a)$. However there is no resolution step issued from $P(c(b), c(f(b)), a)$, then $P(c(b), c(f(b)), a) \notin Mod(Prog)$.

[5] Since $\emptyset$ is flat, a flat goal can always be reached, i.e. in some cases $G = \emptyset$.

Unfortunately, for a given finite CS-program, there may be infinitely many critical pairs. In the following section, this problem is illustrated and some syntactical conditions on CS-program are underlined in order to avoid this critical situation.

3.2 Ensuring finitely many critical pairs

The following example illustrates a situation where the number of critical pairs is unbounded.

▶ **Example 17.** Let $\Sigma = \{f^{\setminus 2}, c^{\setminus 1}, d^{\setminus 1}, s^{\setminus 1}, a^{\setminus 0}\}$ and $f(c(x), y) \to d(y)$ be a rewrite rule, and $Prog = \{P_0(f(x,y)) \leftarrow P_1(x,y).\ P_1(x, s(y)) \leftarrow P_1(x,y).\ P_1(c(x), y) \leftarrow P_2(x,y).\ P_2(a,a) \leftarrow .\}$. Then $P_0(f(c(x), y)) \to P_1(c(x), y) \rightsquigarrow_{y/s(y)} P_1(c(x), y) \rightsquigarrow_{y/s(y)} \cdots P_1(c(x), y) \to P_2(x, y)$. Resolution is applied only on non-flat atoms and the last atom obtained by this derivation is flat. The composition of substitutions along this derivation gives $y/s^n(y)$ for some $n \in \mathbb{N}$. There are infinitely many such derivations, which generates infinitely many critical pairs of the form $P_0(d(s^n(y))) \leftarrow P_2(x, y)$.

This is annoying since the completion process presented in the following needs to compute all critical pairs. This is why we define sufficient conditions to ensure that a given finite CS-program has finitely many critical pairs.

▶ **Definition 18.** $Prog$ is *empty-recursive* if there exist a predicate P and distinct variables $x_1, \ldots, x_n$ s.t. $P(x_1, \ldots, x_n) \rightsquigarrow_\sigma^+ A_1, \ldots, P(x'_1, \ldots, x'_n), \ldots, A_k$ where $x'_1, \ldots, x'_n$ are variables and there exist i, j s.t. $x'_i = \sigma(x_i)$ and $\sigma(x_j)$ is not a variable and $x'_j \in Var(\sigma(x_j))$.

▶ **Example 19.** Let $Prog$ be the CS-program defined as follows:
$Prog = \{P(x', s(y')) \leftarrow P(x', y').\ P(a, b) \leftarrow .\}$
From $P(x, y)$, one can obtained the following derivation: $P(x, y) \rightsquigarrow_{[x/x',\, y/s(y')]} P(x', y')$. Consequently, $Prog$ is empty-recursive since $\sigma = [x/x',\, y/s(y')]$, $x' = \sigma(x)$ and y' is a variable of $\sigma(y) = s(y')$.

The following lemma shows that the non empty-recursiveness of a CS-program is sufficient to ensure the finiteness of the number of critical pairs.

▶ **Lemma 20.** *Let $Prog$ be a normalized CS-program.*
If $Prog$ is not empty-recursive, then the number of critical pairs is finite.

▶ Remark. Note that the CS-program of Example 17 is normalized and has infinitely many critical pairs, however it is empty-recursive because $P_1(x, y) \rightsquigarrow_{[x/x',\, y/s(y')]} P_1(x', y')$.

Proof. By contrapositive. Let us suppose there exist infinitely many critical pairs. So there exist P_1 and infinitely many derivations of the form $(i) : P_1(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \rightsquigarrow_\alpha^* G' \rightsquigarrow_\theta G$ (the number of steps is not bounded). As the number of predicates is finite and every predicate has a fixed arity, there exists a predicate P_2 and a derivation of the form $(ii) : P_2(t_1, \ldots, t_p) \rightsquigarrow_\sigma^k G''_1, P_2(t'_1, \ldots, t'_p), G''_2$ (with $k > 0$) included in some derivation of (i), strictly before the last step, such that :

1. G''_1 and G''_2 are flat.
2. σ is not empty and there exists a variable x in $P_2(t_1, \ldots, t_p)$ such that $\sigma(x) = t$ and t is not a variable and contains a variable y that occurs in $P_2(t'_1, \ldots, t'_p)$. Otherwise we could not have an infinite number of σ necessary to obtain infinitely many critical pairs.

3. At least one term t'_j ($j \in \{1,\ldots,p\}$) is not a variable (only the last step of the initial derivation produces a flat goal G). As we use a CS-clause in each derivation step, we can assume that t'_j is a term among $t_1,\ldots,t_n$ and moreover that $t'_j = t_j$. This property does not necessarily hold as soon as P_2 is reached within (ii). We may have to consider further occurrences of P_2 so that each required term occurs in the required argument, which will necessarily happen because there are only finitely many permutations. So, for each variable x occurring in the non-variable terms, we have $\sigma(x) = x$.
4. From the previous item, we deduce that the variable x found in item 2 is one of the terms $t_1,\ldots,t_p$, say t_k. We can assume that y is t'_k.

If in the (ii) derivation we replace all non-variable terms by new variables, we obtain a new derivation : (iii) : $P_2(x_1,\ldots,x_p) \rightsquigarrow^k_\sigma G''_1, P_2(x'_1,\ldots,x'_p), G''_2$ and there exists i, k such that $\sigma(x_i) = x'_i$ (at least one non-variable term in the (ii) derivation), $\sigma(x_k) = t_k$, and x'_k is a variable of t_k. We conclude that Prog is empty-recursive. ◀

Deciding the empty-recursiveness of a CS-program seems to be a difficult problem (undecidable?). Nevertheless, we propose a sufficient syntactic condition to ensure that a CS-program is not empty-recursive.

▶ **Definition 21.** The clause $P(t_1,\ldots,t_n) \leftarrow A_1,\ldots,Q(\ldots),\ldots,A_m$ is *pseudo-empty over* Q if there exist i,j s.t.

- t_i is a variable,
- and t_j is not a variable,
- and $\exists x \in Var(t_j)$, $x \neq t_i \wedge \{x,t_i\} \subseteq Var(Q(\ldots))$.

Roughly speaking, when making a resolution step issued from the flat atom $P(y_1,\ldots,y_n)$, the variable y_i is not instantiated, and y_j is instantiated by something that is synchronized with y_i (in $Q(\ldots)$).

The clause $H \leftarrow B$ is *pseudo-empty* if there exists some Q s.t. $H \leftarrow B$ is pseudo-empty over Q.

The CS-clause $P(t_1,\ldots,t_n) \leftarrow A_1,\ldots,Q(x_1,\ldots,x_k),\ldots,A_m$ is *empty over* Q if for all x_i, $(\exists j,\ t_j = x_i$ or $x_i \notin Var(P(t_1,\ldots,t_n)))$.

▶ **Example 22.** The CS-clause $P(x,f(x),z) \leftarrow Q(x,z)$ is both pseudo-empty (thanks to the second and the third argument of P) and empty over Q (thanks to the first and the third argument of P).

▶ **Definition 23.** Using Definition 21, let us define two relations over predicate symbols.

- $P_1 \trianglerighteq_{Prog} P_2$ if there exists in $Prog$ a clause empty over P_2 of the form $P_1(\ldots) \leftarrow A_1,\ldots,P_2(\ldots),\ldots,A_n$. The reflexive-transitive closure of $\trianglerighteq_{Prog}$ is denoted by $\trianglerighteq^*_{Prog}$.
- $P_1 >_{Prog} P_2$ if there exist in $Prog$ predicates P'_1, P'_2 s.t. $P_1 \trianglerighteq^*_{Prog} P'_1$ and $P'_2 \trianglerighteq^*_{Prog} P_2$, and a clause pseudo-empty over P'_2 of the form $P'_1(\ldots) \leftarrow A_1,\ldots,P'_2(\ldots),\ldots,A_n$. The transitive closure of $>_{Prog}$ is denoted by $>^+_{Prog}$.

$>_{Prog}$ is *cyclic* if there exists a predicate P s.t. $P >^+_{Prog} P$.

▶ **Example 24.** Let $\Sigma = \{f^{\setminus 1}, h^{\setminus 1}, a^{\setminus 0}\}$ Let $Prog$ be the following CS-program:

$$Prog = \{P(x,h(y),f(z)) \leftarrow Q(x,z),R(y).\ \ Q(x,g(y,z)) \leftarrow P(x,y,z).\ \ R(a) \leftarrow.\ \ Q(a,a) \leftarrow.\}$$

One has $P >_{Prog} Q$ and $Q >_{Prog} P$. Thus, $>_{Prog}$ is cyclic.

The lack of cycles is the key point of our technique since it ensures the finiteness of the number of critical pairs.

▶ **Lemma 25.** *If $>_{Prog}$ is not cyclic, then $Prog$ is not empty-recursive, consequently the number of critical pairs is finite.*

Proof. By contrapositive. Let us suppose that $Prog$ is empty recursive. So there exist P and distinct variables $x_1, \ldots, x_n$ s.t. $P(x_1, \ldots, x_n) \rightsquigarrow_\sigma^+ A_1, \ldots, P(x'_1, \ldots, x'_n), \ldots, A_k$ where $x'_1, \ldots, x'_n$ are variables and there exist i, j s.t. $x'_i = \sigma(x_i)$ and $\sigma(x_j)$ is not a variable and $x'_j \in Var(\sigma(x_j))$. We can extract from the previous derivation the following derivation which has p steps ($p \geq 1$). $P(x_1, \ldots, x_n) = Q^0(x_1, \ldots, x_n) \rightsquigarrow_{\alpha_1} B_1^1 \ldots Q^1(x_1^1, \ldots, x_{n_1}^1) \ldots B_{k_1}^1 \rightsquigarrow_{\alpha_2} B_1^1 \ldots B_1^2 \ldots Q^2(x_1^2, \ldots, x_{n_2}^2) \ldots B_{k_2}^2 \ldots B_{k_1}^1 \rightsquigarrow_{\alpha_3} \ldots \rightsquigarrow_{\alpha_p} B_1^1 \ldots B_1^p \ldots Q^p(x_1^p, \ldots, x_{n_p}^p) \ldots B_{k_p}^p \ldots B_{k_1}^1$ where $Q^p(x_1^p, \ldots, x_{n_p}^p) = P(x'_1, \ldots, x'_n)$.

For each k, $\alpha_k \circ \alpha_{k-1} \ldots \circ \alpha_1(x_i)$ is a variable of $Q^k(x_1^k, \ldots, x_{n_k}^k)$ and $\alpha_k \circ \alpha_{k-1} \ldots \circ \alpha_1(x_j)$ is either a variable of $Q^k(x_1^k, \ldots, x_{n_k}^k)$ or a non-variable term containing a variable of $Q^k(x_1^k, \ldots, x_{n_k}^k)$.

Each derivation step issued from Q^k uses either a clause pseudo-empty over Q^{k+1} and we deduce $Q^k >_{Prog} Q^{k+1}$, or an empty clause over Q^{k+1} and we deduce $Q^k \trianglerighteq_{Prog} Q^{k+1}$. At least one step uses a pseudo-empty clause otherwise no variable from $x_1, \ldots, x_n$ would be instantiated by a non-variable term containing at least one variable in $x'_1, \ldots, x'_n$. We conclude that $P = Q^0\ op_1\ Q^1\ op_2\ Q^2 \ldots Q^{p-1}\ op_p\ Q^p = P$ with each op_i is $>_{Prog}$ or $\trianglerighteq_{Prog}$ and there exists k such that op_k is $>_{Prog}$. Therefore $P >_{Prog}^+ P$, so $>_{Prog}$ is cyclic. ◀

So, if a CS-program $Prog$ does not involve $>_{Prog}$ to be cyclic, then all is fine. Otherwise, we have to transform $Prog$ into another CS-program $Prog'$ such as $>_{Prog'}$ is not cyclic and $Mod(Prog) \subseteq Mod(Prog')$.

The transformation is based on the following observation. If $>_{Prog}$ is cyclic, there is at least one pseudo-empty clause over a given predicate that participates in a cycle. Note that this remark can be checked in Example 24 where $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ is a pseudo-empty clause over Q involving the cycle. To remove cycles, we transform some pseudo-empty clauses into clauses that are not pseudo-empty anymore. It boils down to unsynchronize some variables. The process is mainly described in Definition 28. Definitions 26 and 27 are intermediary definitions involved in Definition 28.

▶ **Definition 26** (simplify)**.** Let $H \leftarrow A_1, \ldots, A_n$ be a CS-clause, and for each i, let us write $A_i = P_i(\ldots)$.
If there exists P_i s.t. $L(P_i) = \emptyset$ then $\mathsf{simplify}(H \leftarrow A_1, \ldots, A_n)$ is the empty set, otherwise it is the set that contains only the clause $H \leftarrow B_1, \ldots, B_m$ such that
- $\{B_i \mid 0 \leq i \leq m\} \subseteq \{A_i \mid 0 \leq i \leq n\}$ and
- $\forall i \in \{1, \ldots, n\}$, $(\neg(\exists j,\ B_j = A_i) \Leftrightarrow Var(A_i) \cap Var(H) = \emptyset)$.

In other words, simplify deletes unproductive clauses, or it removes the atoms of the body that contain only extra-variables.

▶ **Definition 27** (unSync)**.** Let $P(t_1, \ldots, t_n) \leftarrow B$ be a pseudo-empty CS-clause.
$\mathsf{unSync}(P(t_1, \ldots, t_n) \leftarrow B) = \mathsf{simplify}(P(t_1, \ldots, t_n) \leftarrow \sigma_0(B), \sigma_1(B))$ where σ_0, σ_1 are substitutions built as follows:

$$\sigma_0(x) = \begin{cases} x & \text{if } \exists i,\ t_i = x \\ \text{a fresh variable} & \text{otherwise} \end{cases} \qquad \sigma_1(x) = \begin{cases} x & \text{if } \exists i,\ t_i \notin Var \wedge x \in Var(t_i) \\ & \quad \wedge \neg(\exists j,\ t_j = x) \\ \text{a fresh variable} & \text{otherwise} \end{cases}$$

▶ **Definition 28** (removeCycles)**.** Let $Prog$ be a CS-program.

$$\mathsf{removeCycles}(Prog) = \begin{cases} Prog & \text{if } >_{Prog} \text{ is not cyclic} \\ \mathsf{removeCycles}(\{\mathsf{unSync}(H \leftarrow B)\} \cup Prog') & \text{otherwise} \end{cases}$$

where $H \leftarrow B$ is a pseudo-empty clause involved in a cycle and $Prog' = Prog \setminus \{H \leftarrow B\}$.

▶ **Example 29.** Let $Prog$ be the CS-program of Example 24. Since $Prog$ is cyclic, let us compute removeCycles($Prog$). The pseudo-empty CS-clause $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ is involved in the cycle. Consequently, unSync is applied on it. According to Definition 27, one obtains σ_0 and σ_1 where $\sigma_0 = [x/x, y/x_1, z/x_2]$ and $\sigma_1 = [x/x_3, y/y, z/z]$. Thus, one obtains the CS-clause $P(x, h(y), f(z)) \leftarrow Q(x, x_2), R(x_1), Q(x_3, z), R(y)$. Note that according to Definition 27, simplify has to be applied on the CS-clause above-mentioned. Following Definitions 26 and 28, one has to remove $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ from $Prog$ and to add $P(x, h(y), f(z)) \leftarrow Q(x, x_2), Q(x_3, z), R(y)$ instead. Note that the atom $R(x_1)$ has been removed using simplify. Note also that there is no cycle anymore.

Lemma 30 describes that our transformation preserves at least and may extend the initial least Herbrand Model.

▶ **Lemma 30.** *Let $Prog$ be a CS-program and $Prog' =$ removeCycles($Prog$).*
Then $>_{Prog'}$ is not cyclic, and $Mod(Prog) \subseteq Mod(Prog')$. Moreover, if $Prog$ is normalized and preserving, then so is $Prog'$.

Proof. Proof are detailled in [3]. ◀

At this point, given a CS-program $Prog$, if $>_{Prog}$ is not cyclic then the number of critical pairs is finite. Otherwise, we transform $Prog$ into another CS-program $Prog'$ in such a way that $>_{Prog'}$ is not cyclic and $Mod(Prog) \subseteq Mod(Prog')$. Since $Prog'$ is not cyclic, the finiteness of the number of critical pairs is ensured.

3.3 Normalizing critical pairs

In Section 3.1, we have defined the notion of critical pair and we have shown in Theorem 14 that this notion is useful for a matter of rewriting closure. Moreover, as mentioned at the very beginning of Section 3, non-convergent critical pairs correspond to the CS-clauses that we would like to add in the current CS-program. Unfortunately, these CS-clauses are not necessarily in the expected form (normalized).

Definition 34 describes the normalization process that transforms a non-normalized CS-clause into several normalized ones. For example, consider the non-normalized CS-clause $P(f(g(x)), b) \leftarrow P'(x)$. We want to generate a set of normalized CS-clauses covering at least the same Herbrand model. The following set of CS-clauses $\{P(f(x_1), b) \leftarrow P_{new_1}(x_1).\ P_{new_1}(g(x_1)) \leftarrow P'(x_1).\}$ is a good candidate with P_{new_1} a new predicate symbol.

Definition 31 introduces tools for manipulating parameters of predicates (tuple of terms). Definition 32 formalizes a way for cutting a clause head, at depth 1. An example is given after Definition 34.

▶ **Definition 31.** A tree-tuple $(t_1, \ldots, t_n)$ is *normalized* if for all i, t_i is a variable or contains only one function-symbol.
We define tuple concatenation by $(t_1, \ldots, t_n).(s_1, \ldots, s_k) = (t_1, \ldots, t_n, s_1, \ldots, s_k)$.
The arity of the tuple $(t_1, \ldots, t_n)$ is $ar(t_1, \ldots, t_n) = n$.

▶ **Definition 32.** Consider a tree-tuple $\overrightarrow{t} = (t_1, \ldots, t_n)$. We define :

- $\overrightarrow{t}^{cut} = (t_1^{cut}, \ldots, t_n^{cut})$, where $t_i^{cut} = \begin{cases} x'_{i,1} \text{ if } t_i \text{ is a variable} \\ t_i \text{ if } t_i \text{ is a constant} \\ t_i(\epsilon)(x'_{i,1}, \ldots, x'_{i,ar(t_i(\epsilon))}) \text{ otherwise} \end{cases}$

  and variables $x'_{i,k}$ are new variables that do not occur in $\overrightarrow{t}$.

- for each i, $\overrightarrow{Var}(t_i^{cut})$ is the (possibly empty) tuple composed of the variables of t_i^{cut} (taken in the left-right order).
- $\overrightarrow{Var}(\overrightarrow{t}^{cut}) = \overrightarrow{Var}(t_1^{cut}) \dots \overrightarrow{Var}(t_n^{cut})$ (concatenation of tuples).
- for each i, t_i^{rest} is the tree-tuple $t_i^{rest} = \begin{cases} (t_i) \text{ if } t_i \text{ is a variable} \\ \text{the empty tuple if } t_i \text{ is a constant} \\ (t_i|_1, \dots, t_i|_{ar(t_i(\epsilon))}) \text{ otherwise} \end{cases}$
- $\overrightarrow{t}^{rest} = (t_1^{rest} \dots t_n^{rest})$ (concatenation of tuples).

▶ **Example 33.** Let $\overrightarrow{t}$ be a tree-tuple such that $\overrightarrow{t} = (x_1, x_2, g(x_3, h(x_1)), h(x_4), b)$ where x_i's are variables. Thus,

- $\overrightarrow{t}^{cut} = (y_1, y_2, g(y_3, y_4), h(y_5), b)$ with y_i's new variables;
- $\overrightarrow{Var}(\overrightarrow{t}^{cut}) = (y_1, y_2, y_3, y_4, y_5)$;
- $\overrightarrow{t}^{rest} = (x_1, x_2, x_3, h(x_1), x_4)$.

Note that $\overrightarrow{t}^{cut}$ is normalized, $\overrightarrow{Var}(\overrightarrow{t}^{cut})$ is linear, $\overrightarrow{Var}(\overrightarrow{t}^{cut})$ and $\overrightarrow{t}^{rest}$ have the same arity.

Notation: $card(S)$ denotes the number of elements of the finite set S.

▶ **Definition 34** (norm)**.** Let $Prog$ be a normalized CS-program.
Let $Pred$ be the set of predicate symbols of $Prog$, and for each positive integer i, let $Pred_i = \{P \in Pred \mid ar(P) = i\}$ where ar means *arity*.
Let *arity-limit* and *predicate-limit* be positive integers s.t. $\forall P \in Pred$, $arity(P) \leq$ *arity-limit*, and $\forall i \in \{1, \dots,$ *arity-limit*$\}$, $card(Pred_i) \leq$ *predicate-limit*. Let $H \leftarrow B$ be a CS-clause.
Function $\mathsf{norm}_{Prog}(H \leftarrow B)$
Res = Prog
If $H \leftarrow B$ is normalized
then Res = Res $\cup \{H \leftarrow B\}$ (a)
else If $H \rightarrow_{Res} A$ by a synchronizing and non-empty clause
  then (note that A is an atom) Res = $\mathsf{norm}_{Res}(A \leftarrow B)$ (b)
  else let us write $H = P(\overrightarrow{t})$
    If $ar(\overrightarrow{Var}(\overrightarrow{t}^{cut})) \leq$ *arity-limit*
    then let c' be the clause $P(\overrightarrow{t}^{cut}) \leftarrow P'(\overrightarrow{Var}(\overrightarrow{t}^{cut}))$
      where P' is a new or an existing predicate symbol[6]
      $Res = \mathsf{norm}_{Res \cup \{c'\}}(P'(\overrightarrow{t}^{rest}) \leftarrow B)$ (c)
    else choose tuples $\overrightarrow{vt_1}, \dots, \overrightarrow{vt_k}$ and tuples $\overrightarrow{tt_1}, \dots, \overrightarrow{tt_k}$ s.t.
      $\overrightarrow{vt_1} \dots \overrightarrow{vt_k} = \overrightarrow{Var}(\overrightarrow{t}^{cut})$ and $\overrightarrow{tt_1} \dots \overrightarrow{tt_k} = \overrightarrow{t}^{rest}$,
      and for all j, $ar(\overrightarrow{vt_j}) = ar(\overrightarrow{tt_j})$ and $ar(\overrightarrow{vt_j}) \leq$ *arity-limit*
      let c' be the clause $P(\overrightarrow{t}^{cut}) \leftarrow P'_1(\overrightarrow{vt_1}), \dots, P'_k(\overrightarrow{vt_k})$
        where $P'_1, \dots, P'_k$ are new or existing predicate symbols[7]
      Res = Res $\cup \{c'\}$
      For j=1 to k do $Res = \mathsf{norm}_{Res}(P'_j(\overrightarrow{tt_j}) \leftarrow B)$ **EndFor** (d)
    EndIf
  EndIf
EndIf
return Res

[6] If $card(Pred_{ar(\overrightarrow{Var}(\overrightarrow{t}^{cut}))}(Res)) <$ *predicate-limit*, then P' is new, otherwise P' is arbitrarily chosen in $Pred_{ar(\overrightarrow{Var}(\overrightarrow{t}^{cut}))}(Res)$.
[7] For all j, P'_j is new iff $card(Pred_{ar(\overrightarrow{vt_j})}(Res)) + j - 1 <$ *predicate-limit*.

▶ **Example 35.** Consider the CS-program $Prog =$

$\{P_0(f(x)) \leftarrow P_1(x).\ \ P_1(a) \leftarrow .\ \ P_0(u(x)) \leftarrow P_2(x).\ \ P_2(f(x)) \leftarrow P_3(x).\ \ P_3(v(x,x)) \leftarrow P_1(x).\}$

Let *arity-limit* $= 1$ and *predicate-limit* $= 5$. Let $P_2(u(f(v(x,x)))) \leftarrow P_3(x)$ be a CS-clause to normalize. According to Definition 34, we are not in case (a) nor in (b), we are in case (c). Then, according to Definition 32, $\overrightarrow{u(f(v(x,x)))}^{cut} = u(x_1)$ with x_1 a new variable. Since for now the number of predicates with arity 1 is equal to $4 <$ *predicate-limit*, a new predicate P_4 can be created and then one has to add the CS-clause $P_2(u(x_1)) \leftarrow P_4(x_1)$. Then we have to solve the recursive call $\mathsf{norm}_{Prog \cup \{P_2(u(x_1)) \leftarrow P_4(x_1)\}}(P_4(f(v(x,x))) \leftarrow P_3(x))$. The same process is applied except for the creation of a new predicate, because *predicate-limit* would be exceeded. Consequently, no new predicate with arity 1 can be generated. One has to choose an existing one. Let us try with P_3. So, the CS-clause $P_4(f(x_2)) \leftarrow P_3(x_2)$ is added into $Prog$ (because $\overrightarrow{f(v(x,x))}^{cut} = f(x_2)$) and then, norm is called with the parameter $P_3(v(x,x)) \leftarrow P_3(x)$. Finally, $P_3(v(x,x)) \leftarrow P_3(x)$ is also added into $Prog$ since this clause is already normalized. To summarize, the normalization of the CS-clause $P_2(u(f(v(x,x)))) \leftarrow P_3(x)$ has produced three new clauses, which are $P_2(u(x_1)) \leftarrow P_4(x_1)$, $P_4(f(x_2)) \leftarrow P_3(x_2)$ and $P_3(v(x,x)) \leftarrow P_3(x)$.

Obviously, termination of norm is guaranteed according to Lemma 36.

▶ **Lemma 36.** *Function* norm *always terminates.*

Proof. Consider a run of $\mathsf{norm}_{Prog}(H \leftarrow B)$, and any recursive call $\mathsf{norm}_{Prog'}(H' \leftarrow B')$. We can see that $|H'|_\Sigma < |H|_\Sigma$. Consequently a normalized clause is necessarily reached, and there is no recursive call in this case. ◀

Given a normalized CS-program $Prog$, Theorem 37 raises two important points:
1. given a non-normalized clause $H \leftarrow B$, one obtains $H \rightarrow_{\mathsf{norm}_{Prog}(H \leftarrow B)} B$, and 2. adding the CS-clauses provided by norm into $Prog$ may increase the least Herbrand model of $Prog$.

▶ **Theorem 37.** *Let c be a critical pair in $Prog$. Then c is convergent in $\mathsf{norm}_{Prog}(c)$. Moreover for any CS-clause c', we have $Mod(Prog \cup \{c'\}) \subseteq Mod(\mathsf{norm}_{Prog}(c'))$.*

Proof. The second item of the theorem is a consequence of the first item.
Let us now prove the first item. Let $c = (H \leftarrow B)$ and let us prove that $H \rightarrow^*_{Res} B$. The proof is by induction on recursive calls to Function norm (we write *ind-hyp* for "induction hypothesis"). We consider items (a), (b),... in Definition 34 :

(a) From Lemma 30.
(b) We have $H \rightarrow A \rightarrow^*_{ind-hyp} B$.
(c) $H = P(\overrightarrow{t}) \rightarrow_{c'} P'(\overrightarrow{t^{rest}}) \rightarrow^*_{ind-hyp} B$.
(d) $H = P(\overrightarrow{t}) \rightarrow_{c'} (P'_1(\overrightarrow{tt_1}), \ldots, P'_k(\overrightarrow{tt_k})) \rightarrow^*_{ind-hyp} (B, \ldots, B)$ (up to variable renamings).

◀

3.4 Completion

In Sections 3.1 and 3.3, we have described how to detect critical pairs and how to convert them into normalized clauses. Moreover, in a given finite CS-program the number of critical pairs is finite as shown in Section 3.2. Definition 38 explains precisely our technique for computing over-approximation using a CS-program completion.

▶ **Definition 38** (comp). Let R be a left-linear rewrite system, and $Prog$ be a finite and normalized CS-program s.t.

- $>_{Prog}$ is not cyclic (otherwise apply removeCycles to remove cycles),
- and $\forall P \in Pred$, $arity(P) \leq$ *arity-limit*,
- and $\forall i \in \{1, \ldots,$ *arity-limit*$\}$, $card(Pred_i) \leq$ *predicate-limit*.

where $card(Pred_i)$ is the number of predicate symbols of arity i.

Function $\mathsf{comp}_R(Prog)$
while there exists a non-convergent critical pair $H \leftarrow B$ **do**
Prog = removeCycles($\mathsf{norm}_{Prog}(H \leftarrow B)$)
end while
return Prog

Theorem 39 and Corollary 40 illustrate that our technique leads to a finite CS-program whose least Herbrand model over-approximates the descendants obtained by a left-linear rewrite system R.

▶ **Theorem 39.** *Function* comp *always terminates, and all critical pairs are convergent in* $\mathsf{comp}_R(Prog)$. *Moreover* $Mod(Prog) \subseteq Mod(\mathsf{comp}_R(Prog))$.

Proof. Proofs are detailled in [3]. ◀

Moreover, thanks to Theorem 14, $Mod(\mathsf{comp}_R(Prog))$ is closed under rewriting by R. Then:

▶ **Corollary 40.** *If in addition Prog is preserving,* $R^*(Mod(Prog)) \subseteq Mod(\mathsf{comp}_R(Prog))$.

4 Examples

In this section, our technique is applied on several examples. In Examples 41, 42 and 43, I is the initial set of terms and R is the rewrite system. Moreover, initially, we define a CS-program $Prog$ that generates I.

▶ **Example 41.** In this example, we define Σ as follows: $\Sigma = \{c^{\setminus 2}, a^{\setminus 0}\}$. Let I be the set of terms $I = \{f(t) \mid t \in T_\Sigma\}$. Let R be the rewrite system $R = \{f(x) \to b(x, x)\}$. Obviously, one can easily guess that $R^*(I) = \{b(t, t) \mid t \in T_\Sigma\} \cup I$. Note that $R^*(I)$ is not a regular, nor a context-free language [1, 13].

Initially, $Prog = \{P_0(f(x)) \leftarrow P_1(x).\ \ P_1(c(x, y)) \leftarrow P_1(x), P_1(y).\ \ P_1(a) \leftarrow .\}$. Using our approach, the critical pair $P_0(b(x, x)) \leftarrow P_1(x)$ is detected. This critical pair is already normalized, then it is immediately added into $Prog$. Then, there is no more critical pair and the procedure stops. Note that we get exactly the set of descendants, i.e. $L(P_0) = R^*(I)$. So, given $t, t' \in T_\Sigma$ such that $t \neq t'$, one can show that $b(t, t') \notin R^*(I)$.

The example right above shows that non-context-free descendants can be handled in a conclusive manner with our approach. Such example cannot be handled by [14] in an exact way, because they use context-free languages. Actually, the classes of languages covered by our approach and theirs are in some sense orthogonal. However, the examples below shows that our approach can also be relevant for other problems.

▶ **Example 42.**
Let I be the set of terms $I = \{f(a, a)\}$, and R be the rewrite system $R = \{f(x, y) \to u(f(v(x), w(y)))\}$. Intuitively, the exact set of descendants is $R^*(I) = \{u^n(f(v^n(a), w^n(a))) \mid n \in \mathbb{N}\}$. We define $Prog = \{P_0(f(x, y)) \leftarrow P_1(x), P_1(y).\ \ P_1(a) \leftarrow .\}$. We choose *predicate-limit* $= 4$ and *arity-limit* $= 2$.

First, the following critical pair is detected: $P_0(u(f(v(x),w(y)))) \leftarrow P_1(x), P_1(y)$. According to Definition 34, the normalization of this critical pair produces three new CS-clauses: $P_0(u(x)) \leftarrow P_2(x)$, $P_2(f(x,y)) \leftarrow P_3(x,y)$ and $P_3(v(x),w(y)) \leftarrow P_1(x), P_1(y)$. Adding these three CS-clauses into $Prog$ produces the new critical pair $P_2(u(f(v(x),w(y))))$ $\leftarrow P_3(x,y)$. This critical pair can be normalized without exceeding *predicate-limit*. So, we add: $P_2(u(x)) \leftarrow P_4(x)$. $P_4(f(x,y)) \leftarrow P_5(x,y)$. and $P_5(v(x),w(y)) \leftarrow P_3(x,y)$.

Once again, a new critical pair has been introduced: $P_4(u(f(v(x),w(y)))) \leftarrow P_5(x,y)$. Note that, from now, we are not allowed to introduce any new predicate of arity 1. Let us proceed the normalization of $P_4(u(f(v(x),w(y)))) \leftarrow P_5(x,y)$ step by step. We choose to reuse the predicate P_4. Thus, we first generate the following CS-clause: $P_4(u(x)) \leftarrow P_4(x)$. So, we have to normalize now $P_4(f(v(x),w(y))) \leftarrow P_5(x,y)$. Note that $P_4(f(v(x),w(y))) \rightarrow^+_{Prog} P_3(x,y)$. Consequently, the CS-clause $P_3(x,y) \leftarrow P_5(x,y)$ is added into $Prog$.

Note that there is no critical pair anymore.

To summarize, we obtain the final CS-program $Prog_f$ composed of the following CS-clauses:

$$Prog_f = \left\{ \begin{array}{lll} P_0(f(x,y)) \leftarrow P_1(x), P_1(y). & P_1(a) \leftarrow . & P_0(u(x)) \leftarrow P_2(x) \\ P_2(f(x,y)) \leftarrow P_3(x,y). & P_3(v(x),w(y)) \leftarrow P_1(x), P_1(y). & P_2(u(x)) \leftarrow P_4(x). \\ P_4(f(x,y)) \leftarrow P_5(x,y). & P_5(v(x),w(y)) \leftarrow P_3(x,y). & P_4(u(x)) \leftarrow P_4(x). \\ P_3(x,y) \leftarrow P_5(x,y) & & \end{array} \right\}$$

For $Prog_f$, note that $L(P_0) = \{u^n(f(v^m(a),w^m(a))) \mid n,m \in \mathbb{N}\}$ and $R^*(I) \subseteq L(P_0)$.

In Example 42, the approximation computed is still a non-regular language. Nevertheless, it is a strict over-approximation since a synchronization is broken between the three counters.

Let us also show the application of our technique on an example introduced in [5]. In [5] authors propose an example that cannot be handled by regular approximations. Example 43 shows that this limitation can now be overcome.

▶ **Example 43.** Let I be the set of terms $I = \{f(a,a)\}$ and R be the rewrite system $R = \{f(x,y) \rightarrow f(g(x),g(y)),\ f(g(x),g(y)) \rightarrow f(x,y),\ f(a,g(a)) \rightarrow error\}$. Obviously, $R^*(I) = \{f(g^n(a),g^n(a)) \mid n \in \mathbb{N}\}$. Consequently, *error* is not a reachable term.

We start with the CS-program $Prog = \{P_0(f(x,y)) \leftarrow P_1(x), P_1(y).\ P_1(a) \leftarrow .\}$. After applying Function comp, we obtain the following CS-program for any *predicate-limit* ≥ 2:

$$Prog_f = \left\{ \begin{array}{lll} P_0(f(x,y)) \leftarrow P_1(x), P_1(y). & P_0(f(x,y)) \leftarrow P_2(x,y) & P_1(a) \leftarrow . \\ P_2(g(x),g(y)) \leftarrow P_1(x), P_1(y). & P_2(g(x),g(y)) \leftarrow P_2(x,y). & \end{array} \right\}$$

Note that $L(P_0)$ is exactly $R^*(I)$. Note also that $error \notin L(P_0)$. Consequently, we have proved that *error* is not reachable from I.

5 Further Work

We have presented a procedure that always terminates and that computes an over-approximation of the set of descendants, expressed by a synchronized tree language. This is the first attempt using synchronized tree languages. It could be improved or extended:

- In Definition 34, when *predicate-limit* is reached (in items (c) and (d)), an (several in item (d)) existing predicate of the right arity is chosen arbitrarily and re-used, instead of creating a new one. Of course, if there are several existing predicates of the right arity, the achieved choice affects the quality of the approximation. When using regular languages [8], a similar difficulty happens: to make the procedure terminate, it is sometimes necessary to chose and re-use an existing state instead of creating a new one. Some ideas have been proposed to make this choice in a smart way [11]. We are going to extend these ideas in order to improve the choice of existing predicates.

- A similar problem arises when *arity-limit* is reached (item (d)): a tuple is divided into several smaller tuples in an arbitrary way, and there may be several possibilities, which may affect the quality of the approximation.
- To compute descendants, we have used synchronized tree languages, whereas context-free languages have been used in [14]. Each approach has advantages and drawbacks. Therefore, it would be interesting to mix the two approaches to get the advantages of both.

References

1 A. Arnold and M. Dauchet. Un théorème de duplications pour les forêts algébriques. In *Journal of Computer and System Sciences*, pages 13:223–244, 1976.

2 Y. Boichut, B. Boyer, Th. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM*, volume 7635 of *LNCS*, pages 299–315. Springer, 2012.

3 Y. Boichut, J. Chabin, and P. Rety. Over-approximating Descendants by Synchronized Tree Languages. Technical Report RR-2013-04, University of Orléans, LIFO, 2013.

4 Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Finer is Better: Abstraction Refinement for Rewriting Approximations. In *RTA*, volume 5117 of *LNCS*, pages 48–62. Springer, 2008.

5 Y. Boichut and P.-C. Héam. A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. *Information Processing Letters*, 108(1):1–2, 2008.

6 A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular (Tree) Model Checking. *Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

7 M. Dauchet. Simulation of Turing Machines by a Left-Linear Rewrite Rule. In *RTA*, volume 355 of *LNCS*, pages 109–120. Springer, 1989.

8 Th. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *RTA*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.

9 Th. Genet, Th. P. Jensen, V. Kodati, and D. Pichardie. A Java Card CAP Converter in PVS. *ENTCS*, 82(2):426–442, 2003.

10 Th. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *CADE*, volume 1831 of *LNAI*, pages 271–290. Springer-Verlag, 2000.

11 Th. Genet and V. Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with Timbuk. In *LPAR*, volume 2250 of *LNCS*, pages 695–706. Springer, 2001.

12 V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. In *FoSSaCS*, volume 2030 of *LNCS*, pages 214–229. Springer, 2001.

13 D. Hofbauer, M. Huber, and G. Kucherov. Some Results on Top-context-free Tree Languages. In *CAAP*, volume 787 of *LNCS*, pages 157–171. Springer-Verlag, 1994.

14 J. Kochems and C.-H. Luke Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *RTA*, volume 10 of *LIPIcs*, pages 187–202, 2011.

15 S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science*, 1(1):69–98, 1997.

16 S. Limet and G. Salzer. Proving Properties of Term Rewrite Systems via Logic Programs. In *RTA*, volume 3091 of *LNCS*, pages 170–184. Springer, 2004.

17 Sébastien Limet and Gernot Salzer. Tree Tuple Languages from the Logic Programming Point of View. *Journal of Automated Reasoning*, 37(4):323–349, 2006.

Unifying Nominal Unification*

Christophe Calvès

Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
CNRS, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France
Inria, Villers-lès-Nancy, F-54600, France
christophe.calves@gmail.com

Abstract

Nominal unification is proven to be quadratic in time and space. It was so by two different approaches, both inspired by the Paterson-Wegman linear unification algorithm, but dramatically different in the way nominal and first-order constraints are dealt with.

To handle nominal constraints, Levy and Villaret introduced the notion of replacing while Calvès and Fernández use permutations and sets of atoms. To deal with structural constraints, the former use multi-equations in a way similar to the Martelli-Montanari algorithm while the later mimic Paterson-Wegman.

In this paper we abstract over these two approaches and genralize them into the notion of modality, highlighting the general ideas behind nominal unification. We show that replacings and environments are in fact isomorphic. This isomorphism is of prime importance to prove intricate properties on both sides and a step further to the real complexity of nominal unification.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases unification, nominal, α-equivalence, term-graph rewriting

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.143

1 Introduction

Operators with binding are ubiquitous in computer science. Programs, logic formulas, types and proofs are some examples of systems that involve binding. Program transformations, optimisations and compilation, for instance, are defined as operations on programs and should work uniformly on α-equivalence classes. Manipulating terms up to α-equivalence is not easy [8]. Gabbay and Pitts introduced nominal syntax [7, 11] to represent, in a simple and natural way, systems that include binders by extending first-order syntax to provide support for binding operators.

The nominal approach to the representation of systems with binders is characterised by the distinction, at the syntactical level, between atoms (or object-level variables), which can be abstracted (we use the notation $a.t$, where a is an atom and t is a term), and meta-variables (or just variables), which behave like first-order variables but may be decorated with atom permutations. Permutations are generated using swappings (e.g., $(a\ b)\cdot t$ means swap a and b everywhere in t). For instance, $(a\ b)\cdot\lambda a.a = \lambda b.b$, and $(a\ b)\cdot\lambda a.X = \lambda b.(a\ b)\cdot X$ where λ is here a function symbol (we will introduce the notation formally in the next section). As shown in this example, permutations suspend on variables. The idea is that when a substitution is applied to X in $(a\ b)\cdot X$, the permutation will be applied to the term that

* A version including every proof is available online at http://christophe.calves.me

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 143–157
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

instantiates X. Permutations of atoms are one of the main ingredients in the definition of α-equivalence for nominal terms.

Urban, Pitts and Gabbay [13] showed that *nominal unification*, i.e. unification up to α-equivalence, is decidable. They gave an algorithm to find the most general solution to a nominal unification problem, if one exists. A naive implementation, representing terms as trees, is exponential. Cheney proved that a more general form, called *equivariant unification*, is NP-complete [5]. Fortunately nominal unification was proven to be polynomial [5] and even quadratic [9] using reduction to *higher-order patterns* [12].

Levy and Villaret [10], and Calvès and Fernández [1, 2] presented, independently, two very different algorithms to solve nominal unification in quadratic time and space. Both were inspired by the Paterson-Wegman first-order unification algorithm. But they dramatically differ in the way nominal constraints and equations are dealt with: Levy and Villaret use *replacings* (i.e. sequences of abstractions) and multi-equations rewriting system while Calvès and Fernández use *environments* (i.e. permutations and sets of atoms). While being two very different structures, environments and replacings share the same goal: representing constraints generated by abstractions. The actual complexity of nominal unification is still unknown. Could another representation for nominal or first-order constraints lead to a more efficient algorithm? To answer this question we need to abstract over technical details such as representation of these constraints.

Our constributions are:

- We show that the algorithms in [10] and [1, 2] can be unified. The result is a general abstract nominal-unification algorithm that unifies any algorithm based on Paterson-Wegman first-order linear unification and nominal constraints.
- We develop a general notion of nominal modality to enable reasoning about any data structure representing nominal constraints. We showed that the unification algorithm actually relies on four modality operations, so does its complexity.
- We prove that these strutures are isomorphic. So in particular, environments and replacings are. This also means that any representation can be used to establish properties on any modality. We used this to exchange properties between replacings and environments.

The paper is structured as follows. Section 2 presents nominal terms. Section 3 introduces the notion of *nominal modality* as an abstraction of any data structure used to represent nominal constraints. Section 4 defines operations on modalities. Section 5 establishes properties on any modality. Then section 7 presents the unification algorithm on any modality. Section 8 is about the complexity of modalities' operations. Section 9 discusses related and future work. Finally section 10 concludes.

2 Background

Let Σ be a denumerable set of **function symbols** f, g, ...; $\mathcal{X}$ a denumerable set of **variables** $X, Y, \ldots$; and $\mathcal{A}$ a denumerable set of **atoms** $a, b, c, d \ldots$ We assume that these sets are pairwise disjoint. In the intended applications, variables will be used to denote meta-level variables (unknowns), and atoms will be used to represent object-level variables, which can be bound (for instance, atoms may be used to represent the variables of a programming language on which we want to reason).

Let Π denote the set of *finite* permutations over $\mathcal{A}$. Its elements are written π,π_i,π',... In this paper, we represent permutations by lists of *swappings* $((a_1\ b_1)\ldots(a_n\ b_n))$. The empty list is written $\mathtt{id}$, inversion and composition are written respectively π^{-1} and $\pi \circ \pi'$. The support is $\mathtt{supp}(\pi) = \{a \in \mathcal{A} \mid \pi(a) \neq a\}$.

$$\frac{}{\gamma \vdash a\#b}\,(\#_{ab}) \qquad \frac{a\ \#\ X \in \gamma}{\gamma \vdash a\#X}\,(\#_X)$$

$$\frac{}{\gamma \vdash a\#a.s}\,(\#_{absa}) \qquad \frac{\gamma \vdash a\#s \quad a \neq b}{\gamma \vdash a\#b.s}\,(\#_{absb})$$

$$\frac{\gamma \vdash a\#s_1 \quad \dots \quad \gamma \vdash a\#s_n}{\gamma \vdash a\#f(s_1,\dots,s_n)}\,(\#_f) \qquad \frac{\gamma \vdash \pi^{-1}\cdot a\#X}{\gamma \vdash a\#\pi\cdot X}\,(\#_X)$$

$$\frac{}{\gamma \vdash a \approx a}\,(\approx_a) \qquad \frac{}{\gamma \vdash X \approx X}\,(\approx_X)$$

$$\frac{\gamma \vdash s_1 \approx t_1 \quad \dots \quad \gamma \vdash s_n \approx t_n}{\gamma \vdash f(s_1,\dots,s_n) \approx f(t_1,\dots,t_n)}\,(\approx_f) \qquad \frac{\gamma \vdash ds(\pi,\pi')\#X}{\gamma \vdash \pi\cdot X \approx \pi'\cdot X}\,(\approx_\pi)$$

$$\frac{\gamma \vdash s \approx t}{\gamma \vdash a.s \approx a.t}\,(\approx_{absa}) \qquad \frac{\gamma \vdash s \approx (a\ b)\cdot t \quad a\#t \quad a \neq b}{\gamma \vdash a.s \approx b.t}\,(\approx_{absb})$$

Figure 1 Inductive definition of α-equivalence and freshness relations.

▶ **Definition 1.** The set of *nominal terms*, denoted $\mathcal{N}$, is generated by the grammar

$$s,t,u,v,\dots ::= a \mid \pi\cdot X \mid f(s_1,\dots,s_n) \mid a.s$$

where $a.s$ is an *abstraction* and $\pi\cdot X$ a *suspension*.

For example, $a.a$, $a.f(b,g(a))$ and $a.(X,Y)$ are nominal terms.

The action of a permutation π on a term s, written $\pi\cdot s$, is defined by $\pi\cdot a = \pi(a)$, $\pi\cdot(\pi'\cdot X) = (\pi\circ\pi')\cdot X$, $\pi\cdot a.s = \pi(a).\pi\cdot s$ and $\pi\cdot f(s_1,\dots,s_n) = f(\pi\cdot s_1,\dots,\pi\cdot s_n)$.

▶ **Definition 2** (Subterms)**.** Let s and t be two nominal terms. $s \preccurlyeq_{\mathcal{N}} t$ means that s is a subterm of t (possibly t itself). $s \prec_{\mathcal{N}} t$ means that s is a strict subterm of t.

Nominal constraints have the form $a\#t$ and $s \approx t$ (read "a fresh for t" and "s α-equivalent to t", respectively). A freshness context γ is a set of freshness constraints of the form $a\#X$. We define the validity of constraints under a freshness context γ inductively, by a system of axioms and rules, using $\#$ in the definition of $\approx$ (see figure 1). In this figure, we write $ds(\pi,\pi')\#X$ as an abbreviation for $\{a\#X \mid a \in ds(\pi,\pi')\}$, where $ds(\pi,\pi') = \{a \mid \pi\cdot a \neq \pi'\cdot a\}$ is the set of atoms where π and π' differ (i.e., their difference set). a,b are any pair of distinct atoms. The relation $\approx$ is indeed an equivalence relation (see [13] for more details).

Let $\mathbb{R} = \mathcal{P}(\mathcal{N}^2)$ be the set of binary relations on $\mathcal{N}$ and $\mathcal{R}_\alpha \in \mathbb{R}$ be the α-equivalence relation on $\mathcal{N}$. Let Δ be the set of *finite* sets of atoms. Let δ,δ_i,δ',... denote elements of Δ.

Substitutions are mappings from variables X to nominal terms, written σ,σ',.... Composition of substitutions is written $\sigma\circ\sigma'$. $t\sigma$ represent the term where every variable X in t has been replaced by $\sigma(X)$.

3 Nominal Modality

We can express the relation $f(s_1,\dots,s_n) \approx f(t_1,\dots,t_n)$ in terms of relations $s_i \approx t_i$. But we can not express $a.s \approx b.t$ in terms of $s \approx t$ where a and b are two different atoms. The relation between s and t is not the α-equivalence but another relation, written $s \approx_{(a\leftarrow b)\cdot e} t$.

This section aims to express any α-equivalence between two nominal terms in terms of (modal) relations between their subterms. The unification algorithm relies on the ability to compute (efficiently) such relations. More precisely, these relations form a family and can be indexed. This section studies those indexes.

3.1 Nominal Pre-Modality

▶ **Definition 3.** Let $\mathcal{R}$ be in $\mathbb{R}$ and a, b be two atoms. Let $(a \leftarrow b)\cdot\mathcal{R} \in \mathbb{R}$ be the relation defined by:

$$(a \leftarrow b)\cdot\mathcal{R} = \{(s,t) \in \mathcal{N}^2 \mid (a.s, b.t) \in \mathcal{R}\}$$

▶ Proposition 1. Let $\mathcal{R}$ be in $\mathbb{R}$ and let $(a_i, b_i)_{i\in\{1\dots n\}}$ be a finite sequence of pair of atoms :

$$(a_n \leftarrow b_n)\cdot\ldots(a_1 \leftarrow b_1)\cdot\mathcal{R} = \{(s,t) \in \mathcal{N}^2 \mid (a_1.\ldots a_n.s, b_1.\ldots b_n.t) \in \mathcal{R}\}$$

▶ **Definition 4.** A *nominal pre-modality*, or *pre-modality* for short, is a set M provided with a function $\Phi_M : M \to \mathbb{R}$ such that:

$$\begin{array}{llll} \exists e \in M, & & \Phi_M(e) = \mathcal{R}_\alpha & (e) \\ \forall m \in M\ \forall a, b \in \mathcal{A}, & \exists m' \in M, & \Phi_M(m') = (a \leftarrow b)\cdot\Phi_M(m) & (.) \end{array}$$

For example, $\mathbb{R}$ provided with $\Phi_{\mathbb{R}} = \mathtt{id}$ is a pre-modality.

▶ Remark. In the rest of this paper $\Phi_M(m)$ will be written $\mathcal{R}_m$. Furthermore, for any nominal terms s and t, $(s,t) \in \mathcal{R}_m$ will be written $s \approx_m t$.

▶ **Definition 5.** Let M be a pre-modality. Let $\sim$ be the kernel pair of Φ_M, i.e. the equivalence relation on M defined by: $\forall m, m' \in M, \quad m \sim m' \Leftrightarrow \mathcal{R}_m = \mathcal{R}_{m'}$.
The equivalence class of $m \in M$ is written $\overline{m}$.

By abuse of notation, for any pre-modalities M and M', we say that $(m \in M) \sim (m' \in M')$ if $\mathcal{R}_m = \mathcal{R}_{m'}$.

▶ Proposition 2. Let M be a pre-modality, then so is $M_{/\sim}$ with $\Phi_{(M_{/\sim})} = (\Phi_M)_{/\sim}$. Furthermore, $\Phi_{(M_{/\sim})}$ is an injection.

▶ **Corollary 6.** *There is a unique $e \in M_{/\sim}$ such that $\mathcal{R}_e = \mathcal{R}_\alpha$. e is called the neutral element of $M_{/\sim}$.*

In the rest of the paper, when it is not ambiguous, e will denote the neutral element of the considered pre-modality.

▶ **Corollary 7.** *Let $m \in M_{/\sim}$ and $a, b \in \mathcal{A}$. There is a unique $m' \in M_{/\sim}$ such that $\mathcal{R}_{m'} = (a \leftarrow b)\cdot\mathcal{R}_m$. m' is written $(a \leftarrow b)\cdot m$.*

▶ **Definition 8.** Let M be a pre-modality. $[M]$ is defined as the least subset of $M_{/\sim}$ such that:

- $e \in [M]$
- $\forall a, b \in \mathcal{A}, \quad m \in [M] \Rightarrow (a \leftarrow b)\cdot m \in [M]$

▶ **Corollary 9.** *Let M be a pre-modality. For any $m \in [M]$ there exists a finite sequence of pair of atoms $(a_i, b_i)_{i\in\{1\dots n\}}$ such that $m = (a_1 \leftarrow b_1)\cdot\ldots(a_n \leftarrow b_n)\cdot e$.*

▶ Proposition 3. For any pre-modality M, the restriction $\Phi_{[M]} = (\Phi_{M_{/\sim}})\,|_{[M]} \colon [M] \to [\mathbb{R}]$ is a bijection. Furthermore $\Phi_{[M]}(e) = \mathcal{R}_\alpha$ and $\Phi_{[M]}((a \leftarrow b)\cdot m) = (a \leftarrow b)\cdot\Phi_{[M]}(m)$.

3.2 Nominal Modality

▶ **Definition 10.** A *nominal modality* M, or *modality* for short, is a set provided with a bijection $\Phi_M : M \to [\mathbb{R}]$.

- $e = \Phi_M^{-1}(\mathcal{R}_\alpha)$ is called its neutral element.
- For any $a, b \in \mathcal{A}$ and $m \in M$, $(a \leftarrow b)\cdot m = \Phi_M^{-1}((a \leftarrow b)\cdot\mathcal{R}_m)$.

▶ **Corollary 11.** *Let M be a modality, then M is a pre-modality and $[M] \cong M$. Conversely, let M' be a pre-modality, then $[M']$ is a modality.*

▶ Proposition 4. Let M and M' be two modalities. There exist a *unique* bijection $\phi : M \to M'$ such that:

- $\phi(e_M) = e_{M'}$ where e_M (resp. $e_{M'}$) is the neutral element of M (resp. M').
- $\forall a, b \in \mathcal{A}\ \forall m \in M, \quad \phi((a \leftarrow b)\cdot m) = (a \leftarrow b)\cdot\phi(m)$

Such a bijection is called a *nominal-modality isomorphism*, or *modality isomorphism* for short, and is writen $\Phi_{M\to M'}$.

3.3 Environments

This section shows that the set of environments [3] is a pre-modality. Thus, by isomorhism, properties on environments can be transposed to any modality.

▶ **Definition 12.** An *environment* is a pair (π, δ) of a permutation and a set of atoms. The set of environments $(\Pi \times \Delta)$ is written $\mathcal{E}$.

▶ Proposition 5. $\mathcal{E}$ is a pre-modality provided with the function $\Phi_{\mathcal{E}} : \mathcal{E} \to \mathbb{R}$:

$$\mathcal{R}_{(\pi,\delta)} = \{(s,t) \in \mathcal{N}^2 \mid s \approx \pi\cdot t \wedge \delta \# t\}$$

- The neutral element of $\mathcal{E}_{/\sim}$ is $\overline{(\mathtt{id}, \emptyset)}$.
- for any atoms a and b: $(a \leftarrow b)\cdot\overline{(\pi,\delta)} = \overline{((a\ \pi(b)) \circ \pi, (\delta \cup \{\pi^{-1}(a)\}) \setminus \{b\})}$.

▶ Proposition 6. Let $\xi = (\pi, \delta)$ and $\xi' = (\pi', \delta')$ be two environments:

$$\xi \sim \xi' \Leftrightarrow \begin{cases} \delta = \delta' \\ \mathtt{ds}(\pi, \pi') \subseteq \delta \end{cases}$$

▶ Proposition 7. Let π be a permutation and $\delta = \{a_1, \ldots, a_n\} \in \Delta$ (atoms a_i are considered, without loss of generality, distinct), $a'_i = \pi(a_i)$ and $b \notin \delta$:

$$\forall s, t \in \mathcal{N}, \quad s \approx \pi\cdot t \wedge \delta \# t \Leftrightarrow a'_1.b.\ldots a'_n.b.s \approx \underbrace{b.\ldots b.}_{2n \text{ times}}\pi\cdot t$$

▶ Proposition 8. Let π be a permutation, there exists a finite sequence of pair of atoms $(a_i, b_i)_{i\in\{1\ldots n\}}$ such that: $\forall s, t \in \mathcal{N}, \quad s \approx \pi\cdot t \Leftrightarrow a_1.\ldots a_n.s \approx b_1.\ldots b_n.t$

▶ Proposition 9. For any $\xi = (\pi, \delta) \in \mathcal{E}$ there exists a finite sequence of pair of atoms $(a_i, b_i)_{i\in\{1\ldots n\}}$ such that $\mathcal{R}_{(\pi,\delta)} = (a_1 \leftarrow b_1)\cdot\ldots(a_n \leftarrow b_n)\cdot\mathcal{R}_\alpha$. In other words, $\mathcal{E}_{/\sim} = [\mathcal{E}]$.

Thus, for any (pre-)modality M and $m \in M$, we can define the freshness set of m using the isomorphism between environments and M. Precisely:

▶ **Definition 13.** For any modality M and any $m \in M$, let $\overline{\xi}(m) \in \mathcal{E}_{/\sim}$ be the environment equivalence class defined by $\overline{\xi}(m) = \Phi_{M\to\mathcal{E}_{/\sim}}(m)$. We write $\delta(m)$ for the finite set of atoms and $\Pi(m)$ for the set of permutations such that $\overline{\xi}(m) = \Pi(m) \times \{\delta(m)\}$.

3.4 Simple Replacings

Simple replacings [10], or *replacings* for short, were introduced by Levy and Villaret as a draft for *generalized replacings*. Using them to handle nominal constraints is inefficient in practice but being sequences of abstraction they are useful on the theoritical side.

▶ **Definition 14.** A replacing ℓ is a finite sequence of pair of atoms written $(a_i \leftarrow b_i)_{i\in\{1\dots n\}}$. The set of replacings is written $\mathcal{L}$.

▶ **Definition 15** (Concatenation)**.** Let $\ell = (a_i \leftarrow b_i)_{i\in\{1\dots n\}}$ and $\ell' = (a'_i \leftarrow b'_i)_{i\in\{1\dots n'\}}$ be two replacings. The concatenation of ℓ and ℓ', written $\ell :: \ell'$, is the sequence $(a''_i \leftarrow b''_i)_{i\in\{1\dots(n+n')\}}$ with

$$a''_i \ \texttt{(resp.}\ \ b''_i\texttt{)} = \begin{cases} a_i \ \texttt{(resp.}\ \ b_i\texttt{)} & \text{if } 1 \le i \le n \\ a'_{i-n} \ \texttt{(resp.}\ \ b'_{i-n}\texttt{)} & \text{otherwise} \end{cases}$$

The empty sequence is written ε.

▶ Proposition 10. $\mathcal{L}$ provided with $\Phi_{\mathcal{L}} : \mathcal{L} \to \mathbb{R}$ defined by

$$\Phi_{\mathcal{L}}((a_i \leftarrow b_i)_{i\in\{1\dots n\}}) = \{(s,t) \in \mathcal{N}^2 \mid a_n \dots a_1.s \approx b_n \dots b_1.t\}$$

is a pre-modality. The neutral element of $\mathcal{L}_{/\sim}$ is $\overline{\varepsilon}$ and, for any $\ell \in \mathcal{L}$, $(a \leftarrow b)\cdot\overline{\ell} = \overline{(a \leftarrow b) :: \ell}$.

▶ Proposition 11. For any $\ell = (a_i \leftarrow b_i)_{i\in\{1\dots n\}} \in \mathcal{L}$, $\overline{\ell} = (a_1 \leftarrow b_1)\cdot \dots (a_n \leftarrow b_n)\cdot\overline{\varepsilon}$.
In other words, $\mathcal{L}_{/\sim} = [\mathcal{L}]$.

▶ **Definition 16.** For any modality M and any $m \in M$, we write $\overline{\ell}(m) \in \mathcal{L}_{/\sim}$ the replacing defined by $\overline{\ell}(m) = \Phi_{M\to\mathcal{L}_{/\sim}}(m)$.

4 Operations

This section presents all the operations on modal relations (and their modality conterparts) used in the unification algorithms.

4.1 Transposition

▶ **Definition 17.** Let $\mathcal{R} \in \mathbb{R}$ be a relation. The transpose of $\mathcal{R}$, written ${}^t\mathcal{R}$, is defined by

$${}^t\mathcal{R} = \{(s,t) \in \mathcal{N}^2 \mid (t,s) \in \mathcal{R}\}$$

▶ **Definition 18.** For any modality M and $m \in M$ we can define the transpose of m, written ${}^t m$, as ${}^t m = \Phi_{[\mathbb{R}]\to M}({}^t\mathcal{R}_m)$.

▶ Proposition 12. Let $\ell = (a_i \leftarrow b_i)_{i\in\{1\dots n\}} \in \mathcal{L}$ be a replacing, ${}^t\ell = (b_i \leftarrow a_i)_{i\in\{1\dots n\}}$.

▶ Proposition 13. For any modalities M and M', $\forall m \in M, \quad \Phi_{M\to M'}({}^t m) = {}^t(\Phi_{M\to M'}(m))$.

4.2 Support

▶ **Definition 19.** Let $\xi = (\pi, \delta)$ be an environment. The support of ξ, written, $\texttt{supp}(\xi)$, is defined as $\texttt{supp}(\xi) = (\texttt{id}, \texttt{supp}(\pi) \cup \delta)$.

▶ Proposition 14. Let $\xi = (\pi, \delta)$ and $\xi' = (\pi', \delta')$ be two environments: $\xi \sim \xi' \Rightarrow \texttt{supp}(\xi) = \texttt{supp}(\xi')$.
Thus $\texttt{supp}(\_)$ is well-defined on $\mathcal{E}_{/\sim}$: $\texttt{supp}(\overline{\xi}) = \overline{\texttt{supp}(\xi)}$.

▶ **Definition 20.** For any modality M and $m \in M$ we can define $\mathtt{supp}(m)$ as

$$\mathtt{supp}(m) = \Phi_{\mathcal{E}/\sim \to M}(\mathtt{supp}(\Phi_{M \to \mathcal{E}/\sim}(m)))$$

▶ Proposition 15. For any modalities M and M'

$$\forall m \in M, \quad \Phi_{M \to M'}(\mathtt{supp}(m)) = \mathtt{supp}(\Phi_{M \to M'}(m))$$

4.3 Monoid

▶ **Definition 21** (Environment composition)**.** Let $\xi = (\pi, \delta)$ and $\xi' = (\pi', \delta')$ be two environments. The composition of ξ and ξ', written $\xi \circ \xi'$, is defined as $\xi \circ \xi' = (\pi \circ \pi', \pi'^{-1}(\delta) \cup \delta')$.

▶ Proposition 16. Let $\xi \in \mathcal{E}$ be an environment and $e = (\mathtt{id}, \emptyset)$ the neutral element of $\mathcal{E}$: $\xi \circ e = e \circ \xi = \xi$.

▶ **Definition 22** (Relation composition)**.** Let $\mathcal{R}, \mathcal{R}' \in \mathbb{R}$ be two relations. We define the composition of $\mathcal{R}$ and $\mathcal{R}'$, written $\mathcal{R} \circ \mathcal{R}'$, as

$$\mathcal{R} \circ \mathcal{R}' = \{(s, u) \in \mathcal{N}^2 \mid \exists t \in \mathcal{N}, \quad (s, t) \in \mathcal{R} \wedge (t, u) \in \mathcal{R}'\}$$

▶ Proposition 17. Let $\xi, \xi' \in \mathcal{E}$ be two environments, then $\mathcal{R}_{\xi \circ \xi'} = \mathcal{R}_\xi \circ \mathcal{R}_{\xi'}$.

▶ **Corollary 23.** *Let $\mathcal{R}$ and $\mathcal{R}'$ be two relations in $[\mathbb{R}]$. The following propositions hold:*

- $\mathcal{R} \circ \mathcal{R}_\alpha = \mathcal{R}_\alpha \circ \mathcal{R} = \mathcal{R}$
- $\mathcal{R} \circ \mathcal{R}' \in [\mathbb{R}]$

So $[\mathbb{R}]$ provided with $\mathcal{R}_\alpha$ as its neutral element and the relation composition ($\circ$) as its internal law is a monoid.

▶ **Definition 24.** Any modality M can be given a monoid structure whose neutral element is e_M and whose internal law $(\circ) : M^2 \to M$ is $\forall m, m' \in M, \quad m \circ m' = \Phi_{[\mathbb{R}] \to M}(\mathcal{R}_m \circ \mathcal{R}_{m'})$.

▶ Proposition 18. For any modalities M and M'

$$\forall m, m' \in M, \quad \Phi_{M \to M'}(m \circ m') = \Phi_{M \to M'}(m) \circ \Phi_{M \to M'}(m')$$

4.4 Replacings operations

▶ **Definition 25** (`For`)**.** Let M be a modality and $m \in M$, we can define the set of forbidden atoms of m by $\mathtt{For}(m) = \{a \in \mathcal{A} \mid \neg(a \approx_m a)\}$.

▶ **Definition 26** (`Rew`)**.** Let M be a modality and $m \in M$, we can define the renaming function of m by $\mathtt{Rew}(m) = \{(a \leftarrow b) \in \mathcal{A} \times \mathcal{A} \mid a \neq b \wedge a \approx_m b\}$.

5 Properties

This section presents basic properties on modalities used in the unification algorithm. In the following M denotes a modality and m, m', ... denote elements of M.

5.1 Decomposition

These properties are proven by the isomorphism between M and $\mathcal{E}/\sim$.

▶ Proposition 19. $a.s \approx_m b.t \Leftrightarrow s \approx_{(a \leftarrow b) \cdot m} t$

▶ Proposition 20. $f(s_1, \ldots, s_n) \approx_m f(t_1, \ldots, t_n) \Leftrightarrow s_1 \approx_m t_1 \wedge \cdots \wedge s_n \approx_m t_n$

▶ Proposition 21. $(a\ b) \cdot s \approx_m t \Leftrightarrow s \approx_{((a \leftarrow b) \cdot (b \leftarrow a) \cdot e) \circ m} t$

▶ Proposition 22. $s \approx_m (a\ b) \cdot t \Leftrightarrow s \approx_{m \circ ((a \leftarrow b) \cdot (b \leftarrow a) \cdot e)} t$

5.2 Resolution

Decomposition propositions are able to express relations on nominal terms into relations on their subterms but they are stuck faced to relations such as $X \approx_m X$ or $(s \approx_m t \wedge s \approx_{m'} t)$. This subsection shows how to deal with such cases.

▶ Proposition 23. $s \approx_m s \Leftrightarrow s \approx_{\mathtt{supp}(m)} s$

▶ Proposition 24. $s \approx_m t \wedge t \approx_{m'} u \Leftrightarrow s \approx_m t \wedge s \approx_{m \circ m'} u \Leftrightarrow s \approx_{m \circ m'} u \wedge t \approx_{m'} u$

▶ Proposition 25. $s \approx_m s \wedge s \approx_{m'} t \Leftrightarrow s \approx_{\mathtt{supp}(m) \circ m'} t$

▶ Proposition 26. $s \approx_{{}^t m \circ m} s \Leftrightarrow \delta(m) \# s$

▶ Proposition 27. $m \circ {}^t m \circ m = m$

▶ Proposition 28 (`For`). $\mathtt{For}(m) = \delta(\mathtt{supp}(m))$

▶ Proposition 29 (`Rew`). $\mathtt{Rew}(m) = \{(\pi(b) \leftarrow b) \mid b \in \mathtt{supp}(\pi) \setminus \delta(m)\}$ where $\pi \in \Pi(m)$.

6 Modal Problems

▶ **Definition 27** (Equation)**.** A *solution* of the *equation* $s \approx^?_m t$ is a pair (σ, γ) where σ is a substitution and γ freshness context such that $\gamma \vdash s\sigma \approx_m t\sigma$ holds. Two equations are said to be *equivalent* if they have the same set of solutions.

▶ **Definition 28** (Modal Probem)**.** A *nominal modal problem* $\mathcal{P}$ (or *modal problem* for short) is a set of equations of the form $s \approx^?_m t$. (σ, γ) is a *solution* of $\mathcal{P}$ if it is a solution for any equation $s \approx^?_m t$ in $\mathcal{P}$. Two problems are said to be *equivalent* if they have the same set of solutions.

▶ Remark. Note that $s \approx_m t$ is equivalent to $t \approx_{{}^t m} s$. So, in the following, every predicate, pattern, etc involving an equation $s \approx^?_m t$ also matches the equation $t \approx^?_{{}^t m} s$. For example, $a \approx^?_{(c \leftarrow d) \cdot e} b \in \{b \approx^?_{(d \leftarrow c) \cdot e} a\}$.

▶ **Definition 29** (Substitutions)**.** We write $X \mapsto_m s$ for the elementary substitution defined, when $\delta(m) \# X$, by

$$\begin{array}{ll} \sigma(X) = \pi \cdot s & \text{where } \pi \in \Pi(m) \\ \sigma(Y) = Y & \text{otherwise} \end{array}$$

▶ **Definition 30** (Freshness constraints)**.** We write $m \# X$, when $\mathtt{id} \in \Pi(m)$, for the equation $X \approx^?_m X$. Notice that this is equivalent to $\delta(m) \# X$.

The unification algorithm of section 7.2 produces elementary substitution and freshness constraints. They are elementary parts of the incrementatly computed solution. To ease the reading of the paper we have chosen to write them as part of the problem but the reader should keep in mind that they are not inputs but outputs. The "problem" $\{X \mapsto_m s,\ m' \# Y,\ s \approx^?_m t\}$ actually represents the modal problem $\{s \approx^?_m t\}$ where the algorithm has already output the elementary substitution $X \mapsto_m s$ and the freshness constraint $m' \# Y$.

▶ **Definition 31** (Fail)**.** `fail` is to be considered, in the algorithm, as an exception. It means that the problem does not have any solution.

6.1 Graph Representation

▶ **Definition 32.** The *graph representation* of a problem $\mathcal{P}$, written $\mathcal{G}_\mathcal{P}$, is the directed graph whose

- vertex (also called *node*) set, written $\mathcal{G}_{\mathcal{P}\mathcal{V}}$ is the set of all nominal terms appearing in the problem: $\mathcal{G}_{\mathcal{P}\mathcal{V}} = \{r \in \mathcal{N} \mid \exists s,t,m \quad r \preccurlyeq_\mathcal{N} s \wedge s \approx_m t \in \mathcal{P}\}$.
- edge set, written $\mathcal{G}_{\mathcal{P}\mathcal{E}}$, is $\mathcal{G}_{\mathcal{P}\prec_\mathcal{N}} \cup \mathcal{G}_{\mathcal{P}\approx}$ where
  - $\mathcal{G}_{\mathcal{P}\prec_\mathcal{N}}$ is the set of *termgraph edges*. Let $s \rightarrowtail_\mathcal{N} t$ be such an edge. It means that t is a direct proper subterm of of s, i.e. that either $s = a.t$, $s = f(\dots,t,\dots)$ or $s = (a\ b)\cdot t$. s is called a *parent* of t and t is called a *child* of s. In the algorithm, permutation actions $\pi\cdot s$ are not applied but considered as syntactic constructions. For example, let t be the term $(a\ b)(c\ d)\cdot X$. Its child is $(c\ d)\cdot X$ whose child is X.
  - $\mathcal{G}_{\mathcal{P}\approx}$ is the set of equations $s \approx^?_m t$ considered as edges from s to t labelled by m and called *equivalence edge*: $\mathcal{G}_{\mathcal{P}\approx} = \mathcal{P} \cup \{s \approx^?_{(a\leftarrow b)\cdot(b\leftarrow a)\cdot e} t \mid s,t \in \mathcal{G}_{\mathcal{P}\mathcal{V}} \quad s = (a\ b)\cdot t\}$.

▶ Remark. Let $s = (a\ b)\cdot t$ be a term appearing in $\mathcal{P}$. This relation is not only a term edge but also an equivalence relation so it is also represented as an equivalence edge. These relations form the set $\{s \approx^?_{(a\leftarrow b)\cdot(b\leftarrow a)\cdot e} t \mid s,t \in \mathcal{G}_{\mathcal{P}\mathcal{V}} \quad s = (a\ b)\cdot t\}$ which is contained in $\mathcal{G}_{\mathcal{P}\approx}$.

▶ Remark. Several structurally equal subterms can be represented as several nodes but there must be exactly one node per variable in the problem.

▶ Remark. Note that equivalence of equations $s \approx^?_m t$ and $t \approx^?_{{}^t m} s$ implies that an edge from s to t labelled by m is also an edge from t so s labelled by ${}^t m$. Both are considered to be the same edge.

Term edges are good at representing term sharing, but they have the drawback that t is considered as a subterm of $(a\ b)\cdot t$. So a clycle involving of term edge does not implies that the problem have no solution. For example the cycle

$$X \approx^?_e (a\ b) \circ (a\ b)\cdot X,\ (a\ b) \circ (a\ b)\cdot X \rightarrowtail_\mathcal{N} (a\ b)\cdot X,\ (a\ b)\cdot X \rightarrowtail_\mathcal{N} X$$

is valid. We need a notion of term edges such that an edge from s to t means that t is a subterm of s but s and t can not be equivalent up to a modality:

▶ **Definition 33** (Strict subterm)**.** A *strict term edge* is an edge $s \rightarrowtail_{\not\pi} t$ from s to t such that either $s = a.((a_1\ b_1) \circ \dots \circ (a_n\ b_n)\cdot t)$ or $s = f(\dots,(a_1\ b_1) \circ \dots \circ (a_n\ b_n)\cdot t,\dots)$. s is called a *strict parent* of t.

If the graph representation of a problem has a cycle involving a strict term edge, then the problem has no solution.

▶ **Definition 34** (Path)**.** Let `Path` be a predicate on pairs of nodes. $\texttt{Path}(s,t) = \top$ if there exists a path from s to t. Let `SPath` be a predicate on pairs of nodes. $\texttt{SPath}(s,t) = \top$ if there exists a *strict path* (a path involving a strict term edge) from s to t.

▶ **Definition 35** (Term Root)**.** Given a problem $\mathcal{P}$, a *term root* is a node in $\mathcal{G}_\mathcal{P}$ with no parent. A *strict term root* is a node with no strict parent.

▶ **Definition 36** (Occurences)**.** Given a problem $\mathcal{P}$, the occurence of a node n, written $\texttt{occ}(n)$ is defined as the number of its parents and equivalence arrows involing n:
$\texttt{occ}(n) = |\{t \in \mathcal{G}_{\mathcal{P}\mathcal{V}} \mid t \rightarrowtail_\mathcal{N} n \in \mathcal{G}_{\mathcal{P}\prec_\mathcal{N}}\}| + |\{(s,m) \in \mathcal{N} \times M \mid s \approx_m t \in \mathcal{G}_{\mathcal{P}\approx}\}|$

▶ **Definition 37.** Given a graph representation $\mathcal{G}_\mathcal{P}$. A $\approx$-connected component is a connected component for the graph $(\mathcal{G}_{\mathcal{P}\mathcal{V}}, \mathcal{G}_{\mathcal{P}\approx})$ (same vertices but only equivalence edges).

▶ Remark. $\approx$-connected components represents classes of α-eqvuivalent terms up to a modality.

▶ Remark (Garbage collection). Note that, as a representation of a problem, when a term diseappear from the problem, its node in the graph and all the edges involved disapear too. More precisely, when a term root does not appear in any equivalence edges, this node is garbage collected. All of its children become term roots and may be garbage collected to.

7 Unification Algorithm

Though [10] and [2] both rely on the Paterson-Wegnam first-oder linear algorithm, they use very different approaches. This section shows that even this part of the algorithm can be unified. In addition, thanks to the support operation on modalities, the present algorithm can stay general even when dealing with freshness constraints (see propositions 23 and 26).

7.1 Rules

The following rules never create nominal terms. Instead they rewrite edges of the graph representation of the problem and create elementaty substitutions/freshness constaints.

▶ **Definition 38** (Failure rules).

$$\begin{array}{lcl}
a \approx^?_m f(t_1,\dots,t_n) & \to & \texttt{fail} \\
a \approx^?_m b.t & \to & \texttt{fail} \\
f(s_1,\dots,s_n) \approx^?_m a.t & \to & \texttt{fail} \\
a \approx^?_m b \quad \textbf{when } (a,b) \notin \mathcal{R}_m & \to & \texttt{fail}
\end{array}$$

▶ **Definition 39** (Normalisation rules).

$$\begin{array}{rcll}
s \approx^?_m s & \to & s \approx^?_{\texttt{supp}(m)} s & \textbf{if } \texttt{id} \notin \Pi(m) \\
s \approx^?_m s, s \approx^?_{m'} t & \to & s \approx^?_{m \circ m'} t & \textbf{if } \texttt{id} \in \Pi(m) \\
s \approx^?_m t, s \approx^?_{m'} t & \to & s \approx^?_{m \circ \texttt{supp}({}^t m' \circ m)} t & \textbf{if } m \neq m'
\end{array}$$

▶ **Definition 40** (Top to Bottom rules).

$$\begin{array}{rcll}
(a\ b)\cdot s \approx^?_m t & \to & s \approx^?_{((a \leftarrow b)\cdot(b \leftarrow a)\cdot e)\circ m} t & \\
\\
a \approx^?_m b & \to & & \textbf{if } (a,b) \in \mathcal{R}_m \\
f(s_1,\dots,s_n) \approx^?_m f(t_1,\dots,t_n) & \to & s_1 \approx^?_m t_1, \dots, s_n \approx^?_m t_n & \\
a.s \approx^?_m b.t & \to & s \approx^?_{(a \leftarrow b)\cdot m} t & \\
\\
X \approx^?_m r & \to & X \mapsto_m r,\ r \approx^?_{t_{m \circ m}} r & \textbf{if } \texttt{occ}(X) = 1 \wedge r \neq X \\
X \approx^?_m X & \to & m \# X & \textbf{if } \texttt{occ}(X) = 1 \wedge \texttt{id} \in \Pi(m)
\end{array}$$

▶ **Definition 41** ($\approx$-Component Exploration rule). $s \approx^?_m u,\ u \approx^?_{m'} r \to s \approx^?_{m \circ m'} r,\ u \approx^?_{m'} r$

This is clear that all these rules preserve the set of solution as their left-hand sides are equivalent to their right-hand ones as shown in section 5.

7.2 The Paterson-Wegman Strategy

In this section we unify [10] and [2] as a strategy for the rules of section 7.1.

7.2.1 The Strategy

The strategy explores nodes by traversing $\approx$-connected components and assigning to each node n a component representative ($\texttt{repr}(n)$). A component is reduced if all of its nodes are term roots (top to bottom).

▶ **Definition 42.** Let $\mathcal{P}$ be a problem. For any node n, $\texttt{repr}(n)$ represents, if defined ($\bot$ otherwise), an equivalence edge $n \approx^?_m r$ where r is the representative of the $\approx$-connected component of n. Initially $\texttt{repr}(n) = \bot$ for every n. This defines a function `repr` on nodes treated as a global variable.

We want to reduce first $\approx$-connected components whose nodes are all term roots but looking for one would be inefficient. Instead we process a component until we find a node s which is not a term root. Let p be one of its parents. We need to reduce the component of p first to make s a term root. Any reduction performed on the component of s must wait for the one p to be resolved. We implement this priority system as a representative stack $\mathcal{S}$. Only reduction involving the top element of $\mathcal{S}$ are allowed. This garantees that reductions are performed in the correct order.

▶ **Definition 43.** Let $\mathcal{S}$ be a node stack, treated as a global variable. We define two operations on $\mathcal{S}$: $\texttt{push}(n)$ pushes the node n on the stack and `top` represents its top element (if it exists). Note that when one node disappear from the problem, it is also removed from $\mathcal{S}$.

This strategy performs stateful computations. The output (written $\mathcal{O}$), the representative function `repr` and the representative stack $\mathcal{S}$ are global variables so we need to consider a state as a tupple composed of a problem and all of the global variables involved in the strategy. The state generated by Paterson-Wegman Strategy rules verifies some helpful properties so we only consider values of these variables that can be generated by the rules.

▶ **Definition 44.** An ouput, written $\mathcal{O}$, is a set of elementary substitutions and freshness constraints generated by the Paterson-Wegman Strategy rules.

▶ **Definition 45** (State)**.** The set of states $\mathbb{S}$ is defined as the smallet set of 4-tupple $(\mathcal{P}, \mathcal{O}, \texttt{repr}, \mathcal{S})$ (where $\mathcal{P}$ is a problem, $\mathcal{O}$ is an output, `repr` is a repesentative function and $\mathcal{S}$ is a representative stack) such that :

- for any problem $\mathcal{P}$, the *initial state* is $(\mathcal{P}, \emptyset, (\_ \mapsto \bot), \emptyset) \in \mathbb{S}$.
- if a state $\texttt{st} \in \mathbb{S}$, and $\texttt{st}'$ is obtained by applying one of the Paterson-Wegman Strategy rules on `st`, then $\texttt{st}' \in \mathbb{S}$.

▶ **Definition 46** (Unification Algorithm)**.** Let $\mathcal{P}$ be a problem (the input of the algorithm), take $(\mathcal{P}, \emptyset, (\_ \mapsto \bot), \emptyset)$ as the initial state. Then rewrite it using the following rules until a normal form is reached. If `fail` is raised, the problem is considered to have no solution. Otherwise consider the output of the normal state as the most-general unifier of the input problem.

Note that the following rules do work on states, but, in order to ease the reading of the paper, patterns use the expressions $s \approx^?_m t$ and $\texttt{repr}(s)$ to represent respectively the equivalence edge from s to t labelled by m and the image of s by the representative function. Similarly, right-hand sides use the expressions $X \mapsto_m r$ and $m \# X$ to represent respectively the addition to the output of the elementary substition and freshness constraint. The expression $\texttt{repr}(s) := s \approx^?_m r$ means that the image of s by the representative function is set to to $s \approx^?_m r$.

Failure rules are applied at every edge creation. This is done in constant time. Normalisation rules are applied mostly on `repr` at edge creation:

▶ **Definition 47** (Representative Normalisation rules).

$$\begin{array}{rcll} t \approx^?_m t & \to & t \approx^?_{\mathtt{supp}(m)} t & \textbf{if } \mathtt{id} \notin \Pi(m) \\ s \approx^?_m s,\ s \approx^?_{m'} t & \to & s \approx^?_{m \circ m'} t & \textbf{if } \mathtt{id} \in \Pi(m) \\ s \approx^?_m r,\ \mathtt{repr}(s) & \to & \mathtt{repr}(s) := s \approx^?_{m \circ \mathtt{supp}({}^t m' \circ m)} r & \textbf{if } m \neq m' \end{array}$$

where $\mathtt{repr}(s) = s \approx^?_{m'} r$. Furthermore, if $s \approx^?_{m'} t$ or $s \approx^?_m s$ is $\mathtt{repr}(s)$ (resp. $t \approx^?_m t$) then $\mathtt{repr}(s)$ becomes $s \approx^?_{m \circ m'} t$ (resp. $t \approx^?_{\mathtt{supp}(m)} t$).

The *top to bottom rules* are only applied on equations $\mathtt{repr}(s)$ when $\mathtt{occ}(s) = 1$:

▶ **Definition 48** (Top to Bottom Representative rules). The following rules have to be applied only when the left-hand side is a $\mathtt{repr}(s)$ for some s such that $\mathtt{occ}(s) = 1$ ($\mathtt{repr}(s) \to \ldots$ **if** $\mathtt{occ}(s) = 1$):

$$\begin{array}{rcll} (a\ b)\cdot t \approx^?_m r & \to & t \approx^?_{((a \leftarrow b)\cdot(b \leftarrow a)\cdot e) \circ m} r & \\ \\ a \approx^?_m b & \to & & \textbf{if } (a,b) \in \mathcal{R}_m \\ f(s_1,\ldots,s_n) \approx^?_m f(r_1,\ldots,r_n) & \to & s_1 \approx^?_m r_1, \ldots, s_n \approx^?_m r_n & \\ a.t \approx^?_m b.r & \to & t \approx^?_{(a \leftarrow b)\cdot m} r & \\ \\ X \approx^?_m r & \to & X \mapsto_m r,\ r \approx_{{}^t m \circ m} r & \textbf{if } r \neq X \\ X \approx^?_m X & \to & m \# X & \textbf{if } \mathtt{id} \in \Pi(m) \end{array}$$

where $(a\ b)\cdot t$, a, $f(s_1,\ldots,s_n)$, $a.t$ and X are instances of s and r, b, $f(r_1,\ldots,r_n)$ and $b.r$ are the representative.

The $\approx$-component exploration rule have to be applied only when r is the representative:

▶ **Definition 49** ($\approx$-Component Representative Definition rule).

$$s \approx^?_m u,\ \mathtt{repr}(u) \quad \to \quad \begin{cases} (\mathtt{repr}(s) := s \approx^?_{m \circ m'} r),\ \mathtt{repr}(u) & \textbf{if } \mathtt{repr}(s) = \bot \\ \mathtt{fail} & \textbf{if } \mathtt{repr}(s) = s \approx^?_{m''} r' \wedge r' \neq r \\ s \approx^?_{m \circ m'} r,\ \mathtt{repr}(u) & \textbf{if } \mathtt{repr}(s) = s \approx^?_{m''} r \wedge u \neq r \end{cases}$$

where $\mathtt{repr}(u) = u \approx^?_{m'} r$ and $s \notin \{r, u\}$.

The `fail` correspond to the cyclic occurence checking. It occurs when a node s, whose $\mathtt{repr}(s)$ has already been defined as $s \approx^?_{m_1} r_1$ (which means s is in the $\approx$-connected component of r_1), also appear in the $\approx$-connected component of r_2 with $r_1 \neq r_2$. The way representative are selected and defined by traversing term edges make that putting r_1 and r_2 in the same equivalence class would form a cycle.

Finally, we need a rule to initiate the `repr` propagation:

▶ **Definition 50** (`repr` creation).
- If $\mathcal{S} = \emptyset$, then we select a node r, add $\mathtt{repr}(r) = r \approx^?_e r$ to the problem and push r on top of $\mathcal{S}$. r is selected depending on the existance of such a form: if possible, take r of the form $f(\ldots)$, $a.\_$ or an atom, otherwise, take a variable X as r.
- Let s such that $\mathtt{repr}(s) = s \approx^?_m r$, $r = \mathtt{top}$ and s has a strict parent p. Then add $\mathtt{repr}(p) = p \approx^?_e p$ to the problem, fail if $\mathtt{repr}(p) \neq \bot$, and push p on top of $\mathcal{S}$.

7.2.2 Correctness

To prove the correctness, we need to prove that every Paterson-Wegman Strategy is a equivalence on states (so the set of solutions is preserved through rewriting) and that the representative function and stack detect cycles in the solution's graph (occurs-check) (if `fail` is risen, there is no solution).

▶ Proposition 30. A state is equivalent to the problem

$$\mathcal{P} \cup \{X \approx^?_{(\pi,\emptyset)} s \mid X \mapsto_m s \in \mathcal{O},\ \pi \in \Pi(m)\} \cup \{X \approx^?_m X \mid m \# X \in \mathcal{O}\}$$

By language abuse we call solutions of a state the solutions of its equivalent problem.

▶ Proposition 31. Every Paterson-Wegman Strategy rule transforms a state into an equivalent one or `fail`.

Proof. Every rule that does not raise `fail` is an equivalence on the equivalent problem of a state. ◀

Now we need to prove that when `fail` is raised, then the state does not have any solution.

▶ Proposition 32. Let s be a node such that $\texttt{repr}(s) = s \approx^?_m r$. One step of rewriting either alter the modality m ($\texttt{repr}(s)$ becomes $s \approx^?_{m'} r$) or removes complety s from the problem.

▶ **Corollary 51.** $\mathit{repr}(s) = s \approx^?_m r \Rightarrow \mathit{repr}(r) = r \approx^?_{m'} r \wedge r \in \mathcal{S}$

▶ Proposition 33. $\mathcal{S} = [r_1, \ldots, r_i, \ldots, r_j, \ldots, r_n] \Rightarrow \texttt{SPath}(r_i, r_j)$ where $r_n = \texttt{top}$.

▶ **Corollary 52.** *If* $\mathcal{S} = [r_1, \ldots, r_t, \ldots, r_s, \ldots, r_n]$ *and* $s \approx_m t \in \mathcal{P}$ *with* $\mathit{repr}(s) = s \approx^?_{m_s} r_s$, $\mathit{repr}(t) = t \approx^?_{m_t} r_t$ *and* $r_s \neq r_t$. *Then the problem has no solution.*

▶ Proposition 34. If `fail` is raised on a state, then it has no solution.

▶ Proposition 35. An output $\mathcal{O}$ is the most-general unifier of its equivalent problem

$$\{X \approx^?_{(\pi,\emptyset)} s \mid X \mapsto_m s \in \mathcal{O},\ \pi \in \Pi(m)\} \cup \{X \approx^?_m X \mid m \# X \in \mathcal{O}\}$$

7.2.3 Complexity

This section shows that Paterson-Wegman Strategy rules reach a nornal form in a linear number of steps. We consider here one execution of the algorithm on a problem $\mathcal{P}$. The output, `repr` and $\mathcal{S}$ are treated as global variables.

Formally, let $\mathcal{P}_0$ be a problem. The initial state is $\texttt{st}_0 = (\mathcal{P}_0, \emptyset, (\_ \mapsto \bot), \emptyset)$. Let $[\texttt{st}_0, \texttt{st}_1, \ldots]$ be a sequence of states where $\texttt{st}_{i+1}$ is obtained by applying a Paterson-Wegman Strategy rule on $\texttt{st}_i$.

▶ **Definition 53** (Size)**.** The size of a graph representation $\mathcal{G}_\mathcal{P}$, writen $|\mathcal{G}_\mathcal{P}|$, is defined by

$$|\mathcal{G}_\mathcal{P}| = |\mathcal{G}_{\mathcal{P}\mathcal{V}}| + |\mathcal{G}_{\mathcal{P}\mathcal{E}}|$$

▶ **Definition 54** (Mesure)**.** The mesure of $\mathcal{G}_\mathcal{P}$, written $\mu(\mathcal{G}_\mathcal{P})$, is the sum of

- 1 per node in $\mathcal{G}_{\mathcal{P}\mathcal{V}}$.
- 2 per node whose $\texttt{repr}(s) = \bot$.
- 1 per equivalence edge.
- 1 per equivalence edge $s \approx_m t$ with $\texttt{id} \notin \Pi(m)$.
- 2 per equivalence edge $s \approx_m t$ where $\texttt{repr}(s) \neq s \approx_{m'} t$.
- 3 per term edge.

▶ Proposition 36. $\mu(\mathcal{G}_\mathcal{P}) \leq 3|\mathcal{G}_\mathcal{P}|$

▶ Proposition 37. The strategy reduces $\mathcal{G}_\mathcal{P}$ to a normal form in at most $\mu(\mathcal{G}_\mathcal{P})$ steps of rewriting.

▶ **Definition 55.** A state $(\mathcal{P}, \mathcal{O}, \texttt{repr}, \mathcal{S})$ is said to be in *solved form* either if $\mathcal{P} = \emptyset$ or if `fail` has been raised.

▶ Proposition 38. Normal forms are solved form.

▶ Proposition 39. The algorithm computes the most-general unfier of a problem $\mathcal{P}$ if it exists or raise `fail` in a linear number of rewriting steps.

8 Modal Complexity

The complexity of every rewriting steps depends on the complexity of modal operations which itself depends on the modality used.

Let $\mathcal{P}$ be a problem and $\mathcal{A}_\mathcal{P}$ be the set of atoms appearing in $\mathcal{P}$. The unification algorithm does not introduce any atom. So any modal operation computed by any rewriting step only involves atoms in $\mathcal{A}_\mathcal{P}$. If any modal operation can be computed in at most $\theta(|\mathcal{A}_\mathcal{P}|)$, the complexity of the unification algorithm is at most $\theta(|\mathcal{A}_\mathcal{P}| \times \mu(\mathcal{G}_\mathcal{P}))$.

As proven in [3], modal operation on environments can be computed in $\theta(|\mathcal{A}_\mathcal{P}|)$ using intergers as atoms and arrays as permutations and freshness sets. Thus using environments, the algorithm is quadratic in time.

8.1 Replacings

Computing eagerly replacings would be terribly inefficient. Levy and Villaret avoid this complexity by introducing *generalized replacings* which can be seen as a formulae of modal operations. Using subterm sharing, they get a directed acycling graphs of modal-operation formulas for which they compute the sets $\texttt{For}(g)$ and $\texttt{Rew}(g)$ to determine whether $a \approx_g^? b$ is true or not.

▶ **Definition 56** (Generalized Replacings)**.** The set $\mathcal{GL}$ of *generalized replacings* is the set of terms generated by the grammar: $\mathcal{GL} = e \mid (a \leftarrow b)\cdot\mathcal{GL} \mid {}^t\mathcal{GL} \mid \mathcal{GL} \circ \mathcal{GL} \mid \texttt{supp}(\mathcal{GL})$.

▶ Remark. The definition in [10] does not contain $\texttt{supp}(g)$. Instead, multiple occurences of the same variables are kept in multi-equations.

▶ Proposition 40. $\mathcal{GL}$ is a pre-modality.

▶ Proposition 41. $\mathcal{GL}_{/\sim} = [\mathcal{GL}]$

The size of the acyclic graph representing modal-operation formulae is linear in the size of the input problem $\mathcal{P}$ because a linear number of Paterson-Wegman Strategy rules lead to a normal form and each rule involved a bounded number of modal operations. As proven in [10] we can check in quadratic time if the problem has a solution.

9 Related and Future Work

Cheney [4] proved that higher-order pattern unification reduces to nominal unification. Levy and Villaret [9] proved the opposite side. These two results prove that higher-order pattern and nominal unification are equivalent. It would be interesting to adapt modalities to higher-order patterns.

Permissive nominal syntax [6] is a modification to the original syntax to ease the writing of proofs. Given a term t, we know that there are infinitely many fresh atoms for t, but we need freshness constraints to set them. Permissive nominal syntax encodes directly in a term (on variables) which atoms may occur free or not inside. It would be interesting to investigate if our approach can be adapted to take into account this modification.

10 Conclusion

The unique isomorphism between modalities is a powerful tool to establish properties on a representation. Most of the propositions of section 5 where established by proving them on environments and then transposing to any modality by isomorhpism. Furthermore, the algorithm completely isolates nominal constraints from first-order ones. Even when dealing with freshness constraints thanks to Proposition 26.

References

1 Christophe Calvès. *Complexity and Implementation of Nominal Algorithms.* PhD thesis, King's College of London, 2010.

2 Christophe Calvès and Maribel Fernández. The first-order nominal link. In María Alpuente, editor, *LOPSTR*, volume 6564 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2010.

3 Christophe Calvès and Maribel Fernández. Matching and alpha-equivalence check for nominal terms. *J. Comput. Syst. Sci.*, 76(5):283–301, 2010.

4 J. Cheney. Relating nominal and higher-order pattern unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*, pages 104–119, 2005.

5 James Cheney. Equivariant unification. *J. Autom. Reasoning*, 45(3):267–300, 2010.

6 Gilles Dowek, Murdoch James Gabbay, and Dominic P. Mulligan. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Logic Journal of the IGPL*, 18(6):769–822, 2010.

7 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

8 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

9 Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In Andrei Voronkov, editor, *RTA*, volume 5117 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2008.

10 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

11 Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

12 Zhenyu Qian. Linear unification of higher-order patterns. In Marie-Claude Gaudel and Jean-Pierre Jouannaud, editors, *TAPSOFT*, volume 668 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 1993.

13 Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

Rewriting with Linear Inferences in Propositional Logic

Anupam Das

Department of Computer Science
University of Bath
Bath, UK
a.das@bath.ac.uk

Abstract

Linear inferences are sound implications of propositional logic where each variable appears exactly once in the premiss and conclusion. We consider a specific set of these inferences, MS, first studied by Straßburger, corresponding to the logical rules in deep inference proof theory. Despite previous results characterising the individual rules of MS, we show that there is no polynomial-time characterisation of MS, assuming that integers cannot be factorised in polynomial time.

We also examine the length of rewrite paths in an extended system MSU that also has unit equations, utilising a notion dubbed *trivialisation* to reduce the case with units to the case without, amongst other observations on MS-rewriting and the set of linear inferences in general.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases proof theory, propositional logic, complexity of proofs, deep inference

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.158

1 Introduction

Linear inferences are sound implications of propositional logic where the same variables occur in the premiss and conclusion, and occur exactly once in both. For example,

$$A \wedge B \to A \vee B \quad \text{and} \quad A \wedge (B \vee C) \to (A \wedge B) \vee C$$

The left implication is usually known as *mix*, while the right is logically equivalent to $\wedge, \vee$ introduction rules in Gentzen calculi, and is also known as *switch*. While these two rules have traditionally been at the core of proof theory, the advent of *deep inference* proof theory has triggered the study of an additional rule, *medial*:

$$(A \wedge B) \vee (C \wedge D) \to (A \vee C) \wedge (B \vee D)$$

The motivation to consider such a rule is to obtain *locality* for the contraction rule in proofs, an impossible task in traditional Gentzen systems [2]. In recent years there has been much work on understanding the role of medial in proofs and logic [4] [15] [5] [18]. Most recently, Straßburger commenced a study of it from the point of view of rewriting theory [17].

In proof theory we are interested in derivations from one formula to another, under some set of inference rules. In deep inference these rules operate on formulae as in a rewriting system, i.e. they may be applied anywhere in the formula, not just at the root connective. Two typical questions a proof theorist might ask are the following:

1. Is there a derivation from a formula A to a formula B?
2. What is the complexity of a derivation from A to B?

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 158–173

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


In deep inference systems derivations can be considered as rewrite paths by the inference rules, and in this work we ask these questions particularly for the switch-medial fragment.

In [17] Straßburger considered (1) and gave polynomial-time characterisations of switch and medial individually in terms of *relation webs*, graphs that record certain logical information about a formula. An open problem arising from the work was whether a similar characterisation could be given for the combined switch-medial system. In this work we answer this question negatively, if such a characterisation is to decide (1) in polynomial-time, conditional on the assumption that integer factoring cannot be computed by polynomial-size circuits. Along the way we (essentially) show that proof-search in Frege systems (and so also Gentzen/deep inference systems with cut [5]) can be reduced in polynomial time to the search for switch-medial rewrite paths between formulae, suggesting that a lot of the computational content of deep inference proofs lies in this switch-medial fragment.

With regards to (2), it is well-known that switch-medial derivations have polynomial size, in the absence of units. However this does not remain true when units are added, as is common for deep inference proof systems, even after quotienting the set of formulae by unit-equivalences. We exhibit a specific example of this in Sect. 4.1 where we present a derivation using units that contains exponentially many logically distinct formulae. We show that such derivations can only occur when a variable is *trivialised*, i.e. put in disjunction with $\top$ or conjunction with $\bot$, and give a transformation from any switch-medial derivation with units to one of polynomial-size with same premiss and conclusion.

While this is beyond the scope of the current work, the results given have certain consequences for *atomic flows*, diagrams recording structural changes in a proof [11] [9], essentially a type of trace for rewriting derivations. We do not introduce them here, but will briefly comment on these consequences as remarks in this work.

Finally we consider the set of all linear inferences. From the previous results it can be shown that switch and medial are insufficient to derive every linear inference, assuming $\boldsymbol{coNP \neq NP}$. Straßburger gives an explicit linear inference in [19] on 36 variables that cannot be derived, the smallest known thusfar. We improve this result by constructing a linear inference on 10 variables that cannot be derived by switch and medial, even in the presence of units, and conjecture that this is the minimal such inference.

Since this work is primarily motivated by proof theory, we adopt the notational convention presented in [10] for deep inference proofs or, equivalently, rewriting derivations. The main purpose of this work is to better understand the complexity of the logical fragment of deep inference systems, specifically in answering the two questions above.

Acknowledgements

I would like to thank Lutz Straßburger, Alvin Šipraga and Alessio Guglielmi for all the helpful discussions on this work, and also the anonymous referees for their thorough reviews.

2 Preliminaries

The language of propositional logic consists of countably many atoms a, b, etc. and their duals $\bar{a}, \bar{b}$ etc., units $\top, \bot$ and the connectives $\wedge, \vee$, with their usual interpretations. We also have formula variables, or simply variables, denoted A, B, etc. Terms are defined as follows:

$$t ::= \quad a \quad | \quad A \quad | \quad \top \quad | \quad \bot \quad | \quad (t \wedge t) \quad | \quad [t \vee t]$$

The distinction between brackets $(,)$ and $[,]$ is purely a notational convenience to aid the reader distinguish conjunctions from disjuntions.

Ground terms, i.e. terms free of variables, are called *formulae*, and are denoted α, β etc.

▶ Remark (Negation). Note that we have no symbol for negation in our language. Instead atoms come in pairs with their duals and we can express all of propositional logic using the De Morgan laws to push negation to the atoms.

Consequently all contexts (defined below) are positive, and so the soundness of an inference rule is preserved by applying it anywhere in a term.

▶ **Definition 1** (Contexts). A *context* is a term with a hole, denoted $\{\ \}$, occurring in place of a subterm. We write $\xi\{t\}$ to denote the result of substituting the term t for the hole in $\xi\{\ \}$. Contexts can also have multiple holes, e.g. $\xi\{\ \}\cdots\{\ \}$, defined in the natural way.

We identify inference rules with term rewriting rules, and derivations with rewrite paths. Derivations are considered as objects in proof theory, sometimes themselves subject to rewriting, and so it will be convenient to adopt a notation that allows for this. The notation described below was introduced in [10] as a proof formalism, *open deduction*, but can be thought of as just a convenient notation for term rewriting.

▶ **Definition 2.** Let $\mathcal{R}$ be a term rewriting system for propositional logic, i.e. a set of rewrite rules on terms of propositional logic. We write $\begin{matrix} s \\ \| \mathcal{R} \\ t \end{matrix}$ to denote an $\mathcal{R}$-derivation from a term s to a term t, defined as follows:

- $\rho\,\dfrac{s}{t}$ is an $\mathcal{R}$-derivation from s to t, if $s \to t$ is an instance of a rule $\rho : l \to r$ in $\mathcal{R}$.[1]
- $\begin{pmatrix} s & t \\ \| \mathcal{R} \star & \| \mathcal{R} \\ s' & t' \end{pmatrix}$ is an $\mathcal{R}$-derivation from $(s \star t)$ to $(s' \star t')$, for $\star \in \{\wedge, \vee\}$.
- $\begin{matrix} s \\ \| \mathcal{R} \\ t \\ \| \mathcal{R} \\ u \end{matrix}$ is an $\mathcal{R}$-derivation from s to u.

All rewriting rules operate modulo associativity and commutativity of $\vee, \wedge$ in the usual way. For this reason we often exclude internal brackets of a formula.

Sometimes, if two terms s, t are considered equivalent up to some relation, e.g. associativity and commutativity, we may aid the reader by adding a 'fake' rewrite step: $\begin{matrix} s \\ \cdots \\ t \end{matrix}$.

▶ **Definition 3.** A term is *linear* if no variable occurs more than once. A *(sound) linear inference* is a sound[2] rewrite rule $\rho : l \to r$ where l and r are linear terms on the same variables. We define L as the set of all (sound) linear inferences.

▶ **Definition 4.** We define the system MS to consist of the following rules,

$$\mathsf{M}: \quad (A \wedge B) \vee (C \wedge D) \to [A \vee C] \wedge [B \vee D] \quad , \quad \mathsf{S}: \quad A \wedge [B \vee C] \to (A \wedge B) \vee C$$

The system U consists of rules for both directions of the following equations:

$$A \vee \bot = A \quad , \quad A \wedge \top = A \quad , \quad \bot \wedge \bot = \bot \quad , \quad \top \vee \top = \top$$

The system MSU is defined as $\mathsf{MS} \cup \mathsf{U}$.

---

[1] Note in particular that this is a one-step shallow rewrite step.

[2] A rule is sound if every substitution of formulae for formula variables is sound in propositional logic.

3 Complexity of characterising MS

The motivation behind this section originates from the following result in [17].

▶ **Theorem 5** (Straßburger). *There are polynomial-time criteria deciding whether there is a* S *or* M *rewrite path between two terms.*

In the same work the task of characterising MS was raised as an open problem.

In this section we give a polynomial-time reduction from the problem of finding a Frege proof[3] of a given tautology to the problem of finding a MS-rewrite path between two formulae. Consequently, we deduce that there is no polynomial-time characterisation of MS (and also MSU) under the assumption that integers factoring is outside $P/poly$.

Throughout this section we deal with formulae (i.e. ground terms), which are the natural objects of proof theory, rather than generic terms.

3.1 Reducing proof-search to rewriting in MS

We utilise some results from previous work that is beyond the scope of this paper, and so we state them with references but give no proofs. In particular, we refer to a specific deep inference system KSg, on which more can be found in [3],[5].

▶ Notation. For a formula α let $\alpha^n := \overbrace{\alpha \wedge \cdots \wedge \alpha}^{n}$ and $n \cdot \alpha := \overbrace{\alpha \vee \cdots \vee \alpha}^{n}$.

▶ **Proposition 6** (Jeřábek). *A Frege or Gentzen proof (with cut) of a formula τ can be polynomially transformed to a* KSg*-proof of $\tau \vee (a_1 \wedge \bar{a}_1) \vee \cdots \vee (a_n \wedge \bar{a}_n)$, where a_i are the atoms occurring in τ, and vice-versa.*

Proof. See e.g. [13], [8]. ◀

▶ **Proposition 7.** *A* KSg*-proof of a formula τ can be polynomially transformed to a derivation of the following shape,*

$$\begin{array}{l} \bigwedge_i a_i \vee \bar{a}_i \vee \beta_i \\ \quad \Vert \,\mathsf{MS} \\ \quad \tau' \end{array}$$

where τ' differs from τ only by replacing some atom occurrences a by a disjunction $n \cdot a$.

Proof. See e.g. [3], [7]. ◀

▶ **Lemma 8.** *Given a formula $\bigwedge_i a_i \vee \bar{a}_i \vee \beta_i$ there is a polynomial-size derivation of the following form,*

$$\begin{array}{l} \alpha \\ \quad \Vert \,\mathsf{S} \\ \bigwedge_i k_i \cdot a_i \vee k_i \cdot \bar{a}_i \vee \beta_i \end{array}$$

where α is a valid formula in conjunctive normal form, for some k_i.

[3] A Frege proof is a sequence of formulae where each line follows from some previous lines under modus ponens (from α and $\alpha \supset \beta$ infer β) or is drawn from some complete set of axioms.

Proof. Freely apply the inverse of S to each β_i to obtain a formula β'_i of same size in conjunctive normal form, with disjunctions $\beta'_{i1}, \ldots, \beta'_{ik}$. Construct the following derivations as required:

$$\begin{array}{c} [a_i \vee \bar{a}_i \vee \beta'_{i1}] \wedge \cdots \wedge [a_i \vee \bar{a}_i \vee \beta'_{ik}] \\ \Big\Vert \mathsf{S} \\ \left[\begin{array}{r} \beta'_i \\ k \cdot a_i \vee k \cdot \bar{a}_i \vee \Vert \mathsf{S} \\ \beta_i \end{array}\right] \end{array}$$

Validity follows since each disjunction at the top contains a pair of dual atoms. ◀

▶ **Lemma 9.** *Let α be a valid formula in conjunctive normal form, with at least two conjuncts, such that each atom occurs as many times as its dual. Then there is a polynomial-size derivation of the following shape:*

$$\begin{array}{c} \bigwedge_i a_i \vee \bar{a}_i \\ \Big\Vert \mathsf{MS} \\ \alpha \end{array}$$

Proof. Since α is valid each of its disjunctions must contain a pair of dual atoms. If there are two such pairs in some disjunction then build the following derivation:

$$\mathsf{M}\,\dfrac{\mathsf{2{\cdot}S}\,\dfrac{[a \vee \bar{a}] \wedge [\beta \vee \gamma] \wedge \left[b \vee \bar{b} \vee \alpha\right]}{([a \vee \bar{a}] \wedge \beta) \vee \left(\gamma \wedge \left[b \vee \bar{b}\right]\right)}}{\left[a \vee \bar{a} \vee b \vee \bar{b} \vee \alpha\right] \wedge [\beta \vee \gamma]}$$

Read bottom-up, the number of pairs of dual atoms in the same disjunction has reduced and validity has been preserved, so we can repeatedly apply this construction until there are no disjunctions with two pairs of dual atoms.

Now each disjunction has exactly one pair of dual atoms, so match each other atom in a disjunction with an occurrence of its dual in another disjunction; the matching is bijective by the given condition.

We build the following derivation:

$$\mathsf{M}\,\dfrac{\mathsf{2{\cdot}S}\,\dfrac{\alpha \wedge \beta \wedge [a \vee \bar{a}]}{(\alpha \wedge \bar{a}) \vee (\beta \wedge a)}}{[\alpha \vee a] \wedge (\beta \wedge \bar{a})}$$

Read bottom-up, if a and $\bar{a}$ are a matching pair, the total number of matching pairs in distinct disjunctions has reduced and validity has been preserved, so we can repeatedly apply this construction to obtain a derivation of the required form. ◀

▶ **Theorem 10.** *A Frege proof of a formula τ can be polynomially transformed to a derivation of the following form,*

$$\begin{array}{c} \bigwedge_i [a_i \vee \bar{a}_i]^{n_i} \\ \Big\Vert \mathsf{MS} \\ \tau' \end{array}$$

where τ' is obtained from $\sigma = \tau \vee (a_1 \wedge \bar{a}_1) \vee \cdots \vee (a_n \wedge \bar{a}_n)$, where a_i are the atoms occurring in τ, by replacing each atom occurrence a_i by $k \cdot m_i \cdot a_i$, where m_i is the number of occurrences of $\bar{a}_i$ in σ, k is some fixed global constant and n_i is determined by m_i and k by linearity of MS*, and similarly for dual atoms.*

Proof sketch. Follows from Props. 6, 7 and Lemmata 8, 9, under suitable substitution of disjunctions $l \cdot a$ for an atom a everywhere in a MS-derivation. ◀

▶ **Corollary 11.** *Verifying the validity of a tautology τ can be reduced to determining the existence of a* MS*-rewrite path between two formulae in time polynomial in the size of the smallest Frege proof of τ.*

Proof. The premiss and conclusion of the derivations in Thm. 10 are governed by a single parameter, k. We simply run any algorithm that determines the existence of a MS-rewrite path between two formulae on the premiss and conclusion determined by each value of k, from 1 upwards, until it returns. ◀

▶ Remark. The above results could have equivalently been obtained for MSU, rather than MS, with similar proofs.

3.2 No polynomial-time characterisation for MS

By the corollary above, any polynomial-time characterisation of MS would yield an algorithm verifying any tautology in time polynomial in the size of its smallest Frege proof. The existence of such an algorithm for a proof system, known as *weak automatisability*, was proved to be impossible for Frege systems in [1], conditional on the assumption that integer factoring is outside $P/poly$.

▶ **Definition 12.** A proof system P is *weakly automatisable* if there is a procedure verifying the validity of any tautology τ in time polynomial in the size of the smallest P-proof of τ.

▶ **Theorem 13** (Bonet et al.)**.** *If integer factoring is outside $P/poly$ then Frege is not weakly automatisable.*

▶ **Corollary 14.** *If integer factoring is outside $P/poly$ then there is no polynomial-time characterisation of* MS*.*

▶ Remark. With slight modifications, it follows from the results in this section that atomic flows do not form a proof system, in the sense that they cannot be verified in polynomial-time, unless integer factoring is in $P/poly$. This (conditionally) refutes a conjecture of Guglielmi that atomic flows form a proof system [9].

4 Length of paths with units

In this section we address the complexity of rewriting paths in MSU. The length of MS-paths is well-known to be polynomial, and we give a simple proof below that the length is at most cubic in the size of an input term. Much tighter bounds can be obtained, and this is the subject of ongoing work by Bruscoli, Guglielmi and Straßburger.[4]

It should be pointed out that the general belief that units do not contribute to the complexity of a proof is commonplace in the deep inference community, with some results

[4] Personal correspondence.

as folklore, for example the theorem below. Nonetheless, the technicalities of proving this belief, or even formalising what this means, seems nontrivial to the author and this sentiment is communicated via numerous examples.

▶ **Theorem 15.** MS *has only polynomial-length paths.*

Proof. Let $n(t)$ denote the number of $\wedge$s occurring in a term t, and let $m(t)$ denote the number of pairs of leaves in the term-tree of t whose least common connective is $\wedge$. Clearly each medial step reduces the n-value of a term and each switch step reduces the m-value of a term, while not changing the n-value.

Let M denote the product measure $n \times m : t \mapsto (n(t), m(t))$, then each step of an MS-derivation strictly reduces M. But n is linear in the size of a term and m is quadratic, so an MS-derivation can only contain a cubic number of steps. ◀

The situation becomes more complicated when units are considered. Since the rules of U are bidirectional, cycles can be trivially constructed, yielding infinite rewrite paths. Moreover non-cyclic infinite 'increasing' paths can be constructed:

$$a \quad \to \quad \top \wedge a \quad \to \quad \top \wedge \top \wedge a \quad \to \quad \top \wedge \top \wedge \top \wedge a \quad \to \quad \cdots$$

One approach here would be to conduct rewriting modulo the equational theory generated by U, i.e. consider formulae equivalent up to U-rewriting.[5]

▶ **Definition 16** (Rewriting modulo)**.** Let $\mathcal{R}$ be a rewriting system and $\sim$ an equivalence relation on the terms of $\mathcal{R}$. A derivation in $\mathcal{R}/\sim$ is a sequence,

$$s \sim s_1 \to t_1 \sim s_2 \to t_2 \sim \cdots \to t_k \sim t$$

where each $s_i \to t_i$ is a one-step rewrite in $\mathcal{R}$ and $s_i \not\sim t_i$.

We should note that this is a nonstandard definition of rewriting modulo, since we enforce that each rewriting step is between $\sim$-distinct terms. This condition crucially affects termination of a system, but makes sense in the current setting since the equivalence relations induced by our equations can be checked efficiently.

In any case this approach does not quite work here, since we can still construct cycles when rewriting modulo U. For example the following,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\top}{\top \wedge \top}}{[\top \vee a] \wedge [\top \vee b]}\,\mathsf{M}}{\top \wedge \top \quad \vee \quad \top}\,2\cdot\mathsf{S} \qquad \dfrac{\dfrac{a}{a \wedge \top} \vee \dfrac{b}{b \wedge \top}}{[a \vee \top] \wedge [b \vee \top]}\,\mathsf{M}}{\top \vee \top \vee (a \wedge b)}\,2\cdot\mathsf{S}}{\top \vee (a \wedge b)} \vee (a \wedge b)}{\top \vee (a \wedge b)}$$

is a derivation for a cycle $\top \vee (a \wedge b) \to \cdots \to \top \vee a \vee b \to \cdots \to \top \vee (a \wedge b)$.

[5] We will not address here complexity issues arising from such an approach. There are ways to present such rewritings such that each step can still be checked efficiently [5].

These situations only occur when a subterm appears in conjunction with $\bot$ or disjunction with $\top$, a concept we later define as *trivialisation*. They can be avoided by adding to U the following 'non-linear' equations:

$$A \vee \top = \top \qquad A \wedge \bot = \bot$$

Let us call the resulting system U'. We state the following results, whose proofs appear elsewhere and are not difficult to reconstruct.

▶ **Proposition 17.** *Every term is* U'*-equivalent to a unique unit-free term or* $\top$ *or* $\bot$*.*

Proof. See e.g. [7]. ◀

▶ **Proposition 18.** *If two unit-free formulae are distinct, modulo associativity and commutativity, with each atom occurring at most once, then they compute distinct boolean functions.*

Proof. See e.g. [12]. ◀

From these we can deduce the strong normalisation property.

▶ **Corollary 19.** *Rewriting in* MS/U' *is terminating.*

Proof. Assume the input is a formula (i.e. a ground term) without loss of generality, since MS and U' do not distinguish between atoms and variables. By Props. 17 and 18 it follows that, for each step $\alpha \to \beta$ in a MS/U' derivation, α and β compute distinct boolean functions.

There are 2^n assignments on n atoms, and each boolean function determines a unique set of these assignments. Since rewriting in MS/U' preserves logical implication, any rewrite path determines a strictly decreasing sequence of sets of assignments with respect to $\subset$. ◀

Notice that the complexity bound on termination above is exponential, unlike the unit-free case which is polynomial. Perhaps surprisingly, one cannot do better than this, and we prove this by constructing explicit rewrite-paths of exponential length.

4.1 An exponential-length path in MS/U'

We present a new class of rules, collectively known as *supermix*, that are derivable in MSU and show that one can construct exponential-length paths with it, with exponentially many U'-distinct formulae occurring.

▶ **Definition 20** (Supermix)**.** We define the supermix rules, indexed by n, below:

$$\mathsf{smix} \quad : \quad A \vee \bigwedge_{i=1}^{n} B_i \quad \to \quad A \wedge \bigvee_{i=1}^{n} B_i$$

Each supermix rule is clearly a sound linear inference and, for the special case when $n = 1$, it coincides with the usual mix rule.

The following results aim to prove that supermix is derivable in MSU.

▶ **Lemma 21.** *There is a rewrite path from* $\bot$ *to* $\top$ *in both* M/U *and* S/U*.*

Proof.

$$\mathsf{M}\,\dfrac{\dfrac{\bot}{(\bot \wedge \top) \vee (\bot \wedge \top)}}{[\bot \vee \top] \wedge [\bot \vee \top]} \Big/ \top \qquad , \qquad \mathsf{S}\,\dfrac{\dfrac{\bot}{\bot \wedge [\bot \vee \top]}}{(\bot \wedge \bot) \vee \top} \Big/ \top$$

◀

We will simply write $\dfrac{\bot}{\top}$ if we do not mind which rules are used.

▶ **Lemma 22.** *There is a* MS/U *derivation from* $\bigvee_{i=1}^{n} B_i$ *to* $\top \vee \bigwedge_{i=1}^{n} B_i$.

Proof. We proceed by induction on n.

Base Case: by Lemma 21 we have $\dfrac{\bot}{\top} \vee B$.

Inductive Step: Suppose there are such derivations Φ_r for $r < n$. Define:

$$\Phi_n \quad := \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{B_n}{\top \wedge B_n} \vee \quad \dfrac{\bigvee_{i=1}^{n-1} B_i}{\top \vee \bigwedge_{i=1}^{n-1} B_i}\,\Phi_{n-1}\,\mathsf{MSU}}{\top \wedge \left[\top \vee \bigwedge_{i=1}^{n-1} B_i\right]}}{[\top \vee B_n] \wedge \left[\top \vee \top \vee \bigwedge_{i=1}^{n-1} B_i\right]}\,\mathsf{M}}{\top \vee \top \vee \top}\,2{\cdot}\mathsf{S}}{\top} \vee \left(B_n \wedge \bigwedge_{i=1}^{n-1} B_i\right)$$

◀

▶ **Theorem 23.** *Supermix is derivable in* MS/U.

Proof. Let Φ_n be the derivations constructed in Lemma 22. The derivation is as follows:

$$\dfrac{\dfrac{A \wedge \dfrac{\bigvee_{i=1}^{n} B_i}{\top \vee \bigwedge_{i=1}^{n} B_i}\,\Phi_n\,\mathsf{MSU}}{A \wedge \top}\,\mathsf{S}}{A} \vee \bigwedge_{i=1}^{n} B_i$$

◀

Note that the premiss and conclusion of a supermix step are distinct modulo U′, since they are unit-free and compute distinct boolean functions, and so we can construct an exponential-length path in MS/U′ as follows:

$$\Lambda_1 := a_1 \qquad , \qquad \Lambda_{n+1} := \quad \mathsf{smix}\ \dfrac{\dfrac{\bigwedge_{i=1}^{n} a_i}{a_{n+1} \wedge \Lambda_n}\,\mathsf{smix}}{\bigwedge_{i=1}^{n} a_i} \qquad \dfrac{\bigvee_{i=1}^{n} a_i}{a_{n+1} \vee \Lambda_n}\,\mathsf{smix}\ \bigvee_{i=1}^{n} a_i$$

4.2 Construction of polynomial-length paths

The cause of problems in (complexity of) termination of MSU seems to be the trivialising of atoms and variables in a derivation, by putting them in disjunction with $\top$ or conjunction with $\bot$. We define this property formally in this section and show that, although there are paths of exponential length, any two terms with a MSU-path between them has one of polynomial length. The general idea is to 'push' trivialised atoms and variables to one side and reduce to the unit-free case, before reintroducing the trivialised symbols.

Throughout this section we use 'dotted' steps $\overset{s}{\underset{t}{\cdots}}$ to denote U-steps in a derivation, to help distinguish MS-steps from U-steps. This is technically an overloading of notation, but does not cause any problem since there is a polynomial-size MSU-derivation from s to t just if there is a polynomial-size MS/U-derivation from s to t.

▶ **Definition 24** (Trivialisation)**.** A term is *trivial* if it is a disjunction containing $\top$ or a conjunction containing $\bot$. In a derivation we say that an atom or variable is *trivialised* if at any point it occurs inside a trivial subterm.

▶ **Proposition 25.** *There are polynomial-size derivations* $\dfrac{\xi\{A\}}{A \vee \xi\{\bot\}}\,\|_{\mathsf{SU}}$, $\dfrac{A \wedge \xi\{\top\}}{\xi\{A\}}\,\|_{\mathsf{SU}}$.

Proof. See e.g. [3], [16], [13]. The proof is similar to that of Lemma 27. ◀

▶ **Lemma 26.** *Let* $\dfrac{\xi\{A\}}{\zeta\{A\}}\,\|_{\mathsf{MSU}}$ *be a derivation where* A *is trivialised. Then there is a derivation* $\dfrac{\xi\{\top \vee A\}}{\zeta\{\bot \wedge A\}}\,\|_{\mathsf{MSU}}$ *whose size is at most polynomial in the size of the former derivation.*

Proof. There are two cases. In the first case we transform the derivation as follows,

$$\begin{array}{c}\xi\{A\}\\ \Phi\big\|_{\mathsf{MSU}}\\ \eta\{\top \vee \zeta\{A\}\}\\ \Psi\big\|_{\mathsf{MSU}}\\ \xi'\{A\}\end{array} \quad\to\quad \eta\left\{\begin{array}{c}\xi\{\top \vee A\}\\ \Phi'\big\|_{\mathsf{MSU}}\\ \top \vee \zeta\left\{\begin{array}{c}A\\ \cdots\cdots\\ \top \vee \dfrac{[\top \vee \bot] \wedge A}{\top \vee (\bot \wedge A)}\,\mathsf{S}\\ \cdots\cdots\\ \top \vee (\bot \wedge A)\end{array}\right.\\ \bullet\,\|_{\mathsf{S}}\\ \top \vee \zeta\{\bot \wedge A\}\\ \cdots\cdots\\ \top \vee \zeta\{\bot \wedge A\}\\ \Psi'\big\|_{\mathsf{MSU}}\\ \xi'\{\bot \wedge A\}\end{array}\right\}$$

where Φ', Ψ' are obtained by substituting $\top \vee A$, $\bot \wedge A$ resp. everywhere for A, and the derivation marked $\bullet$ is obtained by Prop. 25. In the second case we transform the derivation

as follows,

$$\begin{array}{c}\xi\{A\}\\ \Phi\big\|\mathsf{MSU}\\ \eta\{\bot\wedge\zeta\{A\}\}\\ \Psi\big\|\mathsf{MSU}\\ \xi'\{A\}\end{array}\quad\to\quad \eta\left\{\begin{array}{c}\dfrac{\dfrac{\bot}{\bot\wedge\bot}\wedge}{\zeta\left\{\top\vee\mathsf{S}\dfrac{\dfrac{A}{[\top\vee\bot]\wedge A}}{\top\vee(\bot\wedge A)}\right\}}\\ \bullet\big\|\mathsf{S}\\ 2{\cdot}\mathsf{S}\dfrac{\top\vee\zeta\{\bot\wedge A\}}{\dfrac{(\bot\wedge\top)\vee(\bot\wedge\zeta\{A\})}{\bot\wedge\zeta\{\bot\wedge A\}}}\end{array}\right\}\qquad \begin{array}{c}\xi\{\top\vee A\}\\ \Phi'\big\|\mathsf{MSU}\\ \zeta\{\bot\wedge A\}\end{array}$$

where Φ', Ψ' are obtained by substituting $\top\vee A$, $\bot\wedge A$ resp. everywhere for A, and the derivation marked $\bullet$ is obtained by Prop. 25. ◀

▶ **Lemma 27.** *There are polynomial-size derivations* $\begin{array}{c}(\bot\wedge A)\vee\xi\{\bot\}\\ \|\mathsf{MU}\\ \xi\{\bot\wedge A\}\end{array}$, $\begin{array}{c}\xi\{\top\vee A\}\\ \|\mathsf{MU}\\ [\top\vee A]\wedge\xi\{\top\}\end{array}$.

Proof. We proceed by induction on the depth of the hole in $\xi\{\ \}$. The base cases are trivial, and we give the inductive steps for the first derivation below,

$$\mathsf{M}\dfrac{\left(\dfrac{\bot}{\bot\wedge\bot}\wedge A\right)\vee(\xi\{\bot\}\wedge B)}{\dfrac{(\bot\wedge A)\vee\xi\{\bot\}}{\xi\{\bot\wedge A\}}\,IH\big\|\mathsf{M}}\qquad\wedge\dfrac{\bot\vee B}{B}\qquad,\qquad \dfrac{\dfrac{(\bot\wedge A)\vee[\xi\{\bot\}\vee B]}{(\bot\wedge A)\vee\xi\{\bot\}\vee B}}{\xi\{\bot\wedge A\}}\,IH\big\|\mathsf{M}$$

where derivations marked IH are obtained by the inductive hypothesis. The second derivation is obtained by duality of the inference rules. ◀

▶ **Lemma 28.** *Every* MSU*-derivation where no atoms or variables occur trivialised can be transformed into an* MS*-derivation with* U*-equivalent premiss and conclusion.*

Proof. We simply reduce every line in the derivation to a unit-free term by U. Since no atoms or variables are trivialised we do not need any rules of $\mathsf{U}'\setminus\mathsf{U}$. We rewrite derivations using the four possible cases below, any other combination of rules with units results in some term in either the premiss or conclusion being trivialised.

$$\mathsf{S}\dfrac{s\wedge[\bot\vee t]}{(s\wedge t)\vee\bot}\ \to\ s\wedge t\qquad \mathsf{S}\dfrac{\top\wedge[s\vee t]}{(\top\wedge s)\vee t}\ \to\ s\vee t$$

$$\mathsf{M}\dfrac{(s\wedge t)\vee(\bot\wedge\bot)}{[s\vee\bot]\wedge[t\vee\bot]}\ \to\ s\wedge t\qquad \mathsf{M}\dfrac{(s\wedge\top)\vee(t\wedge\top)}{[s\vee t]\wedge[\top\vee\top]}\ \to\ s\vee t$$

In particular these rewrite rules operate *anywhere* in a derivation. ◀

▶ **Theorem 29.** *Every* MSU*-derivation can be transformed to one with same premiss and conclusion and whose size is polynomial in the size of its premiss and conclusion.*

Proof. Let Φ be an MSU-derivation. If there are no trivialisations then transform it into an MS-derivation by Lemma 28 which must be of polynomial size by Thm. 15.

Otherwise assume there is a trivialised variable in Φ, say A_1, and transform Φ as follows:

$$\begin{array}{c}\xi\{A_1\}\\ \Phi\,\big\|\,\mathsf{MSU}\\ \zeta\{A_1\}\end{array} \quad\to\quad \begin{array}{c}\xi\{\top\vee A_1\}\\ \Phi'\,\big\|\,\mathsf{MSU}\\ \zeta\{\bot\wedge A_1\}\end{array} \quad\to\quad \begin{array}{c}\xi\{\top\vee\bot\}\\ \Phi_1\,\big\|\,\mathsf{MSU}\\ \zeta\{\bot\wedge\bot\}\end{array}$$

where Φ' is obtained from Φ by Lemma 26 and Φ_1 from Φ' by substituting $\bot$ for every instance of A_1.

Now do the same for Φ_1, and repeat this process until either there are no trivialisations in some Φ_k. (Note that it is not sufficient to just do all the trivialised variables at once, since the transformation above may result in new trivialisations.)

Now by Lemma 28 we can transform Φ_k to an MS-derivation Ψ, with same premiss and conclusion modulo U, which we assume to have polynomial size by Thm. 15.

$$\begin{array}{c}\xi\{\top\vee\bot\}\cdots\{\top\vee\bot\}\\ \Phi_k\,\big\|\,\mathsf{MSU}\\ \zeta\{\bot\wedge\bot\}\cdots\{\bot\wedge\bot\}\end{array} \quad\to\quad \begin{array}{c}\xi\{\top\vee\bot\}\cdots\{\top\vee\bot\}\\ \hdashrule{3em}{0.4pt}{1pt}\\ s\\ \Psi\,\|\,\mathsf{MS}\\ t\\ \hdashrule{3em}{0.4pt}{1pt}\\ \zeta\{\bot\wedge\bot\}\cdots\{\bot\wedge\bot\}\end{array}$$

The complete transformation is as follows,

$$\begin{array}{c}\xi\{A_1\}\cdots\{A_k\}\\ \Phi\,\big\|\,\mathsf{MSU}\\ \zeta\{A_1\}\cdots\{A_k\}\end{array} \quad\to\quad \begin{array}{c}\xi\left\{\dfrac{\dfrac{A_1}{\vdots}}{\mathsf{S}\,\dfrac{[\top\vee\bot]\wedge A_1}{\top\vee(\bot\wedge A_1)}}\right\}\cdots\left\{\dfrac{\dfrac{A_k}{\vdots}}{\mathsf{S}\,\dfrac{[\top\vee\bot]\wedge A_k}{\top\vee(\bot\wedge A_k)}}\right\}\\ \circ\,\big\|\,\mathsf{S}\\ \left[\begin{array}{ll}\xi\{\top\vee\bot\}\cdots\{\top\vee\bot\} & \\ \hdashrule{3em}{0.4pt}{1pt}\\ s & \\ \Psi\,\|\,\mathsf{MS} & \vee(\bot\wedge A_1)\vee\cdots\vee(\bot\wedge A_k)\\ t & \\ \hdashrule{3em}{0.4pt}{1pt}\\ \zeta\{\bot\wedge\bot\}\cdots\{\bot\wedge\bot\} & \end{array}\right]\\ \bullet\,\big\|\,\mathsf{M}\\ \zeta\left\{\dfrac{\dfrac{\bot}{\top}\wedge A_1}{\vdots\;A_1}\right\}\cdots\left\{\dfrac{\dfrac{\bot}{\top}\wedge A_k}{\vdots\;A_k}\right\}\end{array}$$

where the derivations marked $\circ, \bullet$ are obtained by repeatedly applying Lemma 27. ◀

▶ Remark. By the above theorem it follows that any derivation can be transformed to one with the same premiss and conclusion, the same atomic flow and whose size is polynomial in the size of its atomic flow. This is tacitly assumed in some papers where the complexity of proofs is controlled by atomic flows, e.g. [6], [8], albeit never in a critical way.

5 The system L of all linear inferences

In the previous sections we considered the specific rules S and M, due to their importance in proof theory, in particular deep inference. However there are infinitely many other inferences one could consider, and there is good reason to analyse the set of all linear inferences, from the point of view of complexity, due to the following result by Straßburger.

▶ **Proposition 30** (Straßburger)**.** L *is* ***coNP****-complete.*

In this section we present two observations, first on a small linear inference not derivable in MSU, and second an extension of the notion of trivialisation that simplifies any search of new linear inferences.

5.1 A linear inference not derivable in MSU

MSU cannot derive every linear inference. This is immediate from Straßburger's result above, and since the length of paths can be assumed to be polynomial, under the assumption that $\boldsymbol{coNP \neq NP}$. Nonetheless Straßburger has given an explicit linear inference on 36 variables that cannot be derived in MS [19]. Here we give an example on 10 variables, and conjecture that it is the minimal inference not derivable in MS. By observing that there are no trivial atoms, the same result follows for MSU.

▶ **Theorem 31.** *The following is a linear inference that is not derivable in* MS*.*

$$\frac{[A \vee (B \wedge B')] \wedge [(C \wedge C') \vee (D \wedge D')] \wedge [(E \wedge E') \vee F]}{([C \vee E] \wedge [A \vee (C' \wedge E')]) \vee ([(B \wedge D) \vee F] \wedge [B' \vee D'])}$$

Proof. The inference is linear by inspection and its soundness can be checked mechanically. However we give an intuitive argument below, to give an idea of its meaning.

The inference is essentially an encoding of the pigeonhole principle with 3 pigeons and 2 holes. Consider the following grid:

A	B B'
C C'	D D'
E E'	F

The linear inference roughly[6] encodes the following statement,

> *if each row contains a box whose variables are true,*
> *then some column has two boxes with a true variable*

which is clearly a tautology since there are more rows than columns. The use of multiple variables in some boxes is so that repetition of variables is avoided, ensuring linearity.

Using this interpretation, it is clear that any application of switch or medial leading to the conclusion must be from a formula not logically implied by the premiss. This can also be checked mechanically. ◀

▶ **Corollary 32.** *The above inference cannot be derived in* MSU*.*

Proof. If it could then some variable must be trivialised by Lemma 28, meaning we could substitute $\top$ for it in the premiss and $\bot$ in the conclusion and obtain a valid implication. Inspection shows that no variable has this property (the aforementioned interpretation makes it easier to verify this). ◀

[6] Not exactly since not all combinations of variables in boxes are exhausted.

5.2 Towards a basis for L

Can we find a basis for L? I.e. can we find some polynomial-time decidable set of linear inferences from which every linear inference can be derived? This question remains open, but it is worth noting that such a set cannot be finite; the encoding in Thm. 31 can easily be generalised to arbitrary $n \times (n-1)$ grids, and it is not difficult to show that each subsequent linear inference cannot be derived from all the previous ones, along with MSU. It is also worth noting that any basis would have to admit (necessarily) superpolynomial-length paths, unless $\boldsymbol{coNP} = \boldsymbol{NP}$.

Here we present an observation extending the previous notion of trivialisation. We considered previously *syntactic* trivialisation of an atom or variable, when it is explicitly put in disjunction with $\top$ or conjunction with $\bot$. However, when talking about all linear inferences we will want a more general concept that is not reliant on how it is derived in any particular system:

▶ **Definition 33** (Semantic trivialisation)**.** Let $\rho : \xi\{A\} \to \zeta\{A\}$ be a linear inference. We say that ρ is *semantically trivial* at A, or simply trivial, if $\xi\{\top\} \to \zeta\{\bot\}$ is sound.

The condition in the above definition is equivalent to demanding that $\xi\{s\} \to \zeta\{t\}$ is sound for every s, t.

Note that trivialities may depend on each other, and so one should say that an inference is "trivial at A then B" or "trivial at A or B" rather than "trivial at A and B". For example $\mathsf{mix} : A \wedge B \to A \vee B$ is trivial at A or B but not both at once.

▶ **Theorem 34.** *If a linear inference ρ is trivial somewhere then there is a linear inference ρ' on fewer variables that is not trivial anywhere and from which ρ is derivable in* MSU*.*

Proof. Let $\rho : s \to t$ and let $A_1, \ldots, A_k$ be the trivial variables (in order). We construct the following derivation in $\rho' \cup \mathsf{MS/U}$,

$$\dfrac{\dfrac{s}{\xi\left\{A_1 \vee \dfrac{\bot}{\top}\right\}\cdots\left\{A_k \vee \dfrac{\bot}{\top}\right\}}\bullet\Vert\mathsf{M}}{\left(\begin{array}{l}[A_1 \vee \top] \wedge \cdots \wedge [A_k \vee \top] \wedge \dfrac{\dfrac{\xi\{\top\}\cdots\{\top\}}{\rho'\ \dfrac{s'}{t'}}}{\zeta\left\{\dfrac{\bot}{\top \wedge \bot}\right\}\cdots\left\{\dfrac{\bot}{\top \wedge \bot}\right\}}\end{array}\right)}\circ\Vert\mathsf{S}$$
$$\dfrac{\zeta\left\{\mathsf{S}\dfrac{[A_1 \vee \top] \wedge \bot}{A_1 \vee (\top \wedge \bot)}\right\}\cdots\left\{\mathsf{S}\dfrac{[A_k \vee \top] \wedge \bot}{A_k \vee (\top \wedge \bot)}\right\}}{t}$$

where the derivation marked $\bullet$ is obtained from Lemma 27, the derivation marked $\circ$ from Prop. 25 and s', t' are the unique unit-free terms U-equivalent to $\xi\{\top\}\cdots\{\top\}$, $\zeta\{\bot\}\cdots\{\bot\}$ respectively. ◀

6 Conclusions

In this work we considered the linear inferences of propositional logic, in particular from the point of view of complexity and termination of rewriting derivations. This was motivated by the seemingly fundamental role played by linear inferences in deep inference proof theory; as well as being necessary for locality of the inference rules in deep inference, we showed in Sect. 3 that proof search in Frege and Gentzen systems with cut can be reduced in polynomial-time to finding MS-rewrite paths. In contrast, we showed in Sect. 4 that the length of MS(U)-rewrite paths can always be made polynomial, and so the size of a proof is determined by the use of structural rules in a deep inference derivation. Finally we considered the set of all linear inferences and made some general observations.

One particular outcome of this research is the possibility to implement proof search based on strong systems. Typically, proof search algorithms are based on weak proof systems, due to an apparent tradeoff between proof size and proof search. This is most significantly exemplified by the presence of *nonanalytic* rules in stronger systems, e.g.

$$\frac{A \quad A \supset B}{B}\ \textit{modus ponens} \qquad \frac{\Gamma \to \Delta, A \quad A, \Sigma \to \Pi}{\Gamma, \Sigma \to \Delta, \Pi}\ \textit{cut}$$

When searching for a proof we tend to work 'bottom-up', and in the two rules above there are seemingly infinitely many choices for A, which is terrible for proof-search. The tradeoff is that weak systems, such as cut-free Gentzen and Resolution, have much larger proofs. In many cases there are only exponential-size proofs, as opposed to polynomial-size ones in Frege systems [14], for example the propositional encodings of the pigeonhole principle. This lower bound acts as a barrier to efficient proof search, since the complexity of the search procedure is bounded below by the complexity of the objects it searches for.

However, in Sect. 3 we gave a polynomial-time reduction of the problem of proof-search in Frege and Gentzen systems to finding MS-rewrite paths between formulae. This is arguably a simpler problem, firstly since there is no infinite choice present as variables in a term are preserved by linear inferences, and secondly since we already have some understanding of various subproblems, namely a characterisation of S and M in [17]. It would be interesting to see what progress could be made on proof search algorithms based on MS-rewriting, enabling access to the shorter proofs of stronger systems while still restricting the nondeterminism of proof search.

Even more powerful systems, e.g. Extended Frege, could also be used as a base for proof search in the same way, by adding more linear rules. A proof system P can be simulated by Frege when axioms expressing the soundness of P are added [14], and using a trick from [19] these can be encoded as linear inference rules which could be added to MS, again preserving analyticity.

References

1 ML Bonet, T. Pitassi, and R. Raz. No feasible interpolation for tc0-frege proofs. In *focs*, page 254. Published by the IEEE Computer Society, 1997.

2 Kai Brünnler. Two restrictions on contraction. *Logic Journal of the IGPL*, 11(5):525–529, 2003. `http://www.iam.unibe.ch/~kai/Papers/RestContr.pdf`.

3 Kai Brünnler. *Deep Inference and Symmetry in Classical Proofs*. Logos Verlag, Berlin, 2004. `http://www.iam.unibe.ch/~kai/Papers/phd.pdf`.

4 Kai Brünnler and Alwen Fernanto Tiu. A local system for classical logic. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR 2001*, volume 2250 of *Lecture Notes in Computer Science*, pages 347–361. Springer-Verlag, 2001. `http://www.iam.unibe.ch/~kai/Papers/lcl-lpar.pdf`.

5 Paola Bruscoli and Alessio Guglielmi. On the proof complexity of deep inference. *ACM Transactions on Computational Logic*, 10(2):1–34, 2009. Article 14. `http://cs.bath.ac.uk/ag/p/PrComplDI.pdf`.

6 Paola Bruscoli, Alessio Guglielmi, Tom Gundersen, and Michel Parigot. Quasipolynomial normalisation in deep inference via atomic flows and threshold formulae. Submitted. `http://cs.bath.ac.uk/ag/p/QuasiPolNormDI.pdf`, 2009.

7 Anupam Das. On the proof complexity of cut-free bounded deep inference. 2011. Tableaux'11.

8 Anupam Das. Complexity of deep inference via atomic flows. In S. Barry Cooper, Anuj Dawar, and Benedikt Löwe, editors, *Computability in Europe*, volume 7318 of *Lecture Notes in Computer Science*, pages 139–150. Springer-Verlag, 2012. `http://www.anupamdas.com/items/RelComp/RelComp.pdf`.

9 Alessio Guglielmi and Tom Gundersen. Normalisation control in deep inference via atomic flows. *Logical Methods in Computer Science*, 4(1:9):1–36, 2008. `http://www.lmcs-online.org/ojs/viewarticle.php?id=341`.

10 Alessio Guglielmi, Tom Gundersen, and Michel Parigot. A proof calculus which reduces syntactic bureaucracy. In Christopher Lynch, editor, *RTA 2010*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 135–150. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010. `http://drops.dagstuhl.de/opus/volltexte/2010/2649`.

11 Alessio Guglielmi, Tom Gundersen, and Lutz Straßburger. Breaking paths in atomic flows for classical logic. In Jean-Pierre Jouannaud, editor, *25th Annual IEEE Symposium on Logic in Computer Science*, pages 284–293. IEEE, 2010. `http://www.lix.polytechnique.fr/~lutz/papers/AFII.pdf`.

12 VA Gurvich. Repetition-free boolean functions. *Uspekhi Matematicheskikh Nauk*, 32(1):183–184, 1977.

13 Emil Jeřábek. Proof complexity of the cut-free calculus of structures. *Journal of Logic and Computation*, 19(2):323–339, 2009. `http://www.math.cas.cz/~jerabek/papers/cos.pdf`.

14 Jan Krajíček. *Bounded arithmetic, propositional logic, and complexity theory*. Cambridge University Press, New York, NY, USA, 1995.

15 François Lamarche. Exploring the gap between linear and classical logic. *Theory and Applications of Categories*, 18(17):473–535, 2007. `http://www.loria.fr/~lamarche/papers/Gap.pdf`.

16 Lutz Straßburger. MELL in the calculus of structures. *Theoretical Computer Science*, 309:213–285, 2003. `http://www.lix.polytechnique.fr/~lutz/papers/els.pdf`.

17 Lutz Straßburger. A characterisation of medial as rewriting rule. In Franz Baader, editor, *RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 344–358. Springer-Verlag, 2007. `http://www.lix.polytechnique.fr/~lutz/papers/CharMedial.pdf`.

18 Lutz Straßburger. On the axiomatisation of boolean categories with and without medial. *Theory and Applications of Categories*, 18(18):536–601, 2007. `http://www.lix.polytechnique.fr/~lutz/papers/medial.pdf`.

19 Lutz Straßburger. Extension without cut. *Ann. Pure Appl. Logic*, 163(12):1995–2007, 2012.

Proof Orders for Decreasing Diagrams

Bertram Felgenhauer[1] and Vincent van Oostrom[2]

1 **Institute for Computer Science, University of Innsbruck**
`bertram.felgenhauer@uibk.ac.at`
2 **Department of Philosophy, Utrecht University**
`Vincent.vanOostrom@phil.uu.nl`

Abstract

We present and compare some well-founded proof orders for decreasing diagrams. These proof orders order a conversion above another conversion if the latter is obtained by filling any peak in the former by a (locally) decreasing diagram. Therefore each such proof order entails the decreasing diagrams technique for proving confluence. The proof orders differ with respect to monotonicity and complexity. Our results are developed in the setting of involutive monoids. We extend these results to obtain a decreasing diagrams technique for confluence modulo.

1998 ACM Subject Classification F.4 Mathematical Logic and Formal Languages

Keywords and phrases involutive monoid, confluence modulo, decreasing diagram, proof order

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.174

1 Introduction

In this paper we revisit the decreasing diagrams technique [10] for proving confluence. We exhibit several well-founded orders on proofs that allow us to prove termination of the proof transformation system defined by the locally decreasing diagrams. A similar approach is used in the correctness proof for completion by Bachmair and Dershowitz [2]. Rather than working on proofs directly, we develop our orders in the setting of involutive monoids, which capture the essential structure of proofs—proofs may be concatenated and reversed.

This work is partly inspired by [5], where a well-founded order on proofs is defined in order to establish that local decreasingness implies confluence. In [4], a simplified version of this proof order is defined. The orders presented here are much simpler.

The remainder of this paper is structured as follows: In Section 2 we introduce basic notions used in our paper. Section 3 presents involutive monoids. In Section 4 we develop orders on so-called French strings that entail the decreasing diagrams technique. Then, in Section 5, we extend our approach to the Church–Rosser modulo property, using an extension of French strings that we call Greek strings, leading to a generalisation of a results by Ohlebusch [8] and Jouannaud and Liu [4]. Finally, we conclude in Section 6.

Throughout we illustrate our constructions by means of the following running example.

▸ **Example 1.** The rewrite relation $\to$ on objects $\{a, \ldots, j\}$ as presented on the left in Figure 1 is the union of the family of rewrite relations $(\underset{\ell}{\to})_{\ell \in L}$ on its right, indexed by concrete labels $L = \{\ell, m, \kappa\}$ and having individual rewrite relations:

$$\underset{\kappa}{\to} = \{(b,c),(j,i)\}$$
$$\underset{\ell}{\to} = \{(d,c),(f,a),(f,h),(g,e),(h,a),(e,j)\}$$
$$\underset{m}{\to} = \{(b,a),(d,e),(c,f),(c,g),(g,i),(a,i),(h,i)\}$$

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 174–189

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

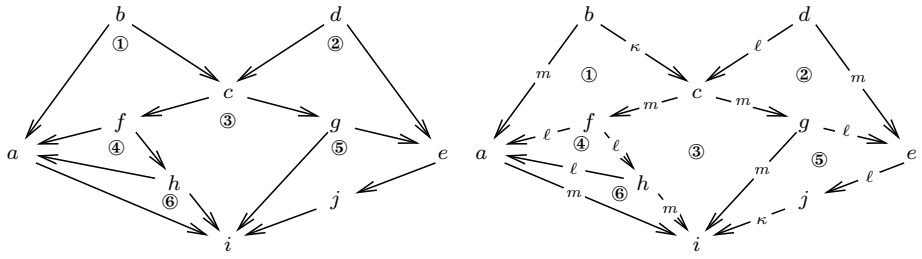


Figure 1 Decomposing a rewrite relation (left) into a family of such (right).

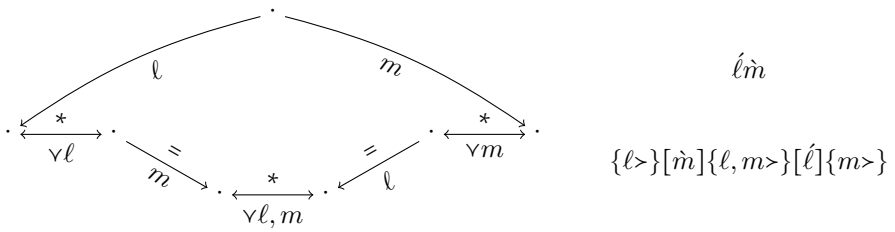


Figure 2 Locally decreasing diagram (left). Interpretations of peak and joining conversion (right).

We will show how each of the transformation steps, indicated by the numbers, leading from the initial conversion $a \underset{m}{\leftarrow} b \underset{\kappa}{\rightarrow} c \underset{\ell}{\leftarrow} d \underset{m}{\rightarrow} e$ to the final valley $a \underset{m}{\rightarrow} i \underset{\kappa}{\leftarrow} j \underset{\ell}{\leftarrow} e$ entails a decrease in each of our proof orders, which are based on some well-founded order $\succ$ on L.

2 Preliminaries

We use standard notions of term rewriting. For a comprehensive overview, see [9].

We use arrow-like notations like $\rightarrow$ for rewrite relations, i.e. binary relations on a set (also called abstract rewrite systems), and symmetric notations like $\mapsto$ for symmetric rewrite relations. The inverse of $\rightarrow$, its reflexive closure, transitive closure, reflexive–transitive closure, and its n-th power of $\rightarrow$ are denoted by $\leftarrow$, $\overset{=}{\rightarrow}$, $\overset{+}{\rightarrow}$, $\overset{*}{\rightarrow}$, and $\overset{n}{\rightarrow}$ respectively. In particular, $\overset{0}{\rightarrow}$ is the identity relation. If $(\underset{\ell}{\rightarrow})_{\ell \in L}$ is a family of rewrite relations and $M \subseteq L$, we let $\underset{M}{\longrightarrow} = \bigcup_{\ell \in M} \underset{\ell}{\rightarrow}$.

Given an alphabet L of labels endowed with a well-founded order (precedence) $\succ$, we define $\curlyvee \ell = \{\kappa \in L \mid \ell \succ \kappa\}$ and $\curlyvee \ell, m = \curlyvee \ell \cup \curlyvee m$. The gist of the decreasing diagrams technique [10] is that to show that a rewrite relation is confluent, we can decompose it into an L-indexed family of rewrite relation $(\underset{\ell}{\rightarrow})_{\ell \in L}$. Then, if every local peak $u \underset{\ell}{\leftarrow} \cdot \underset{m}{\longrightarrow} v$ can be joined decreasingly, that is, there is a joining conversion $u \overset{*}{\underset{\curlyvee \ell}{\longleftrightarrow}} \cdot \overset{=}{\underset{m}{\longrightarrow}} \cdot \overset{*}{\underset{\curlyvee \ell, m}{\longleftrightarrow}} \cdot \overset{=}{\underset{\ell}{\longleftarrow}} \cdot \overset{*}{\underset{\curlyvee m}{\longleftrightarrow}} v$ (see Figure 2, left), the relation $\underset{L}{\longrightarrow}$ is confluent.

Throughout we assume $\succ$ is a well-founded partial order on the labels L.

Rewriting modulo is concerned with pairs of relations $\rightarrow$ and $\mapsto$, where $\mapsto$ is symmetric. Let $\Leftrightarrow = \leftrightarrow \cup \mapsto$. We say that $\rightarrow$ is Church–Rosser modulo $\mapsto$ iff $\overset{*}{\Leftrightarrow} \subseteq \overset{*}{\rightarrow} \cdot \overset{*}{\mapsto} \cdot \overset{*}{\leftarrow}$. In order to apply the decreasing diagrams technique, we distinguish between local peaks $\leftarrow \cdot \rightarrow$ and local cliffs $\leftarrow \cdot \mapsto$ or $\mapsto \cdot \rightarrow$. (There are other notions of confluence for rewriting modulo. See [8] for a systematic discussion.)

$$\frac{t \to u}{t = u}\ \text{(step)} \qquad \frac{}{t = t}\ \text{(reflexive)} \qquad \frac{u = t}{t = u}\ \text{(symmetric)} \qquad \frac{t = u \quad u = v}{t = v}\ \text{(transitive)}$$

Figure 3 Equational logic for rewrite relations.

$$\frac{\dfrac{t = u \quad u = v}{t = v} \quad v = w}{t = w} \xRightarrow{\text{assoc}} \frac{t = u \quad \dfrac{u = v \quad v = w}{u = w}}{t = w} \qquad \frac{\dfrac{t = u \quad u = v}{t = v}}{v = t} \xRightarrow[\text{automorph}]{\text{anti-}} \frac{\dfrac{u = v}{v = u} \quad \dfrac{t = u}{u = t}}{v = t}$$

$$\frac{t = u \quad \dfrac{}{u = u}}{t = u} \xRightarrow{\text{right id}} t = u \qquad \frac{\dfrac{}{t = t} \quad t = u}{t = u} \xRightarrow{\text{left id}} t = u \qquad \frac{\dfrac{t = u}{u = t}}{t = u} \xRightarrow{\text{involl}} t = u \qquad \frac{\dfrac{}{t = t}}{t = t} \xRightarrow{\text{inv id}} \frac{}{t = t}$$

Figure 4 Normalising equational logic proofs into conversions.

3 Involutive monoids

A conversion $t \overset{*}{\leftrightarrow} u$ for a rewrite relation $\to$ is a witness to a proof that t is equal to u in the equational logic induced by $\to$, see Figure 3.

▸ Remark. Because of the absence of term structure the equational logic is particularly simple: terms t,u,v are constants and the usual substitution and congruence rules are superfluous.

However, conversions correspond only to a subset of the equational logic proofs. For example, in a conversion symmetry is never applied below transitivity. In general, conversions can be identified with equational logic proofs that are in normal form with respect to the transformations in Figure 4.[1] Since these transformations are confluent and terminating, every equational logic proof can be transformed into a conversion so one may restrict attention to the latter, a result known as logicality of rewriting with respect to equational logic.

Involutive monoids, see e.g. [3], are the natural algebraic structure to interpret such equational proofs *and* their transformations. In a slogan: involutive monoids are to conversions what monoids are to reductions.[2] More precisely, involutive monoids are obtained by abstracting the equalities into primitives a,b,c,..., interpreting transitivity as composition $(\cdot)$, symmetry as inversion (-1), reflexivity as identity (e), and equipping these with the laws in Definition 2 corresponding to the transformations of Figure 4.

▸ **Definition 2.** A *monoid* is a (*carrier*) set endowed with an associative binary operation $(\cdot)$ and an identity element (e). An *involutive* monoid is a monoid endowed with an anti-automorphic involution (-1), i.e. satisfying the following laws:[3]

$$\begin{array}{rcll} (a \cdot b) \cdot c & = & a \cdot (b \cdot c) & \text{(associative)} \\ a \cdot e & = & a & \text{(right identity)} \\ e \cdot a & = & a & \text{(left identity)} \end{array} \qquad \begin{array}{rcll} (a \cdot b)^{-1} & = & b^{-1} \cdot a^{-1} & \text{(anti-automorphic)} \\ (a^{-1})^{-1} & = & a & \text{(involutive)} \\ e^{-1} & = & e & \text{(inverse identity)} \end{array}$$

Involutive monoids are the main algebraic structure into which conversions and transformations on them will be interpreted, in this paper. This will be the topic of the next section. We

[1] Reductions can be identified with proofs of rewrite logic in normal form.
[2] For *term* rewriting the involutive monoid is to be extended with operations corresponding to the function symbols and laws for them yielding (equational) proof term algebras.
[3] $e^{-1} = e$ is superfluous as it is derivable: $e^{-1} = e \cdot e^{-1} = (e^{-1})^{-1} \cdot e^{-1} = (e \cdot e^{-1})^{-1} = (e^{-1})^{-1} = e$.

now illustrate involutive monoids first by some (mostly well-known) examples from algebra to be used later, and next by our main example, the involutive monoid of French strings.

▸ **Example 3.** (i) The integers with addition, zero, and unary minus $(\mathbb{Z}, +, 0, -)$ constitute an involutive monoid. In general, any group constitutes an involutive monoid;

(ii) The monoid of natural numbers with addition and zero $(\mathbb{N}, +, 0)$ constitute an involutive monoid when endowed with the identity map, as do the multisets over L with multiset sum and the empty multiset $([L], \uplus, [])$. Commutative monoids give rise to involutive monoids in this way;

(iii) (Ordinary) strings over an alphabet L of *labels* or *letters* ℓ, endowed with juxtaposition, the empty string ε, and string reversal constitute an involutive monoid.

(iv) Natural number pairs with pointwise addition, the pair $(0, 0)$, and swapping constitute an involutive monoid. In fact, any monoid $(A, \cdot, e)$ gives rise to an involutive monoid on $A \times A$ by endowing it with pointwise composition, the pair (e, e), and swapping;

(v) Natural number triples endowed with $\cdot$ defined by

$$(n_1, m_1, k_1) \cdot (n_2, m_2, k_2) = (n_1 + n_2, m_1 + k_1 \cdot n_2 + m_2, k_1 + k_2)$$

zero $(0, 0, 0)$, and $(n, m, k)^{-1} = (k, m, n)$, constitute an involutive monoid. In fact, we will only employ triples such that the middle component does not exceed the product of the other components. Such triples can be given a geometric interpretation as diagrams, as illustrated in Figure 5 right, and for this reason we will refer to them as *area* triples.

Our interpretations of conversions with respect to a family $(\underset{\ell}{\to})_{\ell \in L}$ of rewrite relations indexed by labels in L, will all factor through an interpretation (see Definition 8) that only keeps the labels, equipping them with accents according to the direction (forward or backward) of the individual steps in the conversion. We call such strings of accented labels French strings.[4]

▸ **Definition 4.** For a given alphabet L, let a *French* letter be an accented (grave $\grave{\ell}$ or acute $\acute{\ell}$) letter. We will use the circumflex as in $\hat{\ell}$ to denote a French letter having ℓ as label and carrying either a grave or acute accent. The set $\widehat{L}$ of *French* strings over L, i.e. strings of French letters, endowed with juxtaposition, the empty string, and mirroring -1 given by $\grave{\ell}^{-1} = \acute{\ell}$ and $\acute{\ell}^{-1} = \grave{\ell}$, constitute an involutive monoid. The order $\succ$ on labels is extended to French letters: we let $\hat{\ell} \succ \hat{m}$ and $\ell \succ \hat{m}$ iff $\ell \succ m$.

For instance, mirroring the French string $\acute{m}\grave{k}\acute{\ell}\grave{m}$ over the alphabet of Example 1 yields $\acute{m}\grave{\ell}\acute{k}\grave{m}$. In case the alphabet is a singleton set, the French letters over the alphabet are identified with the accents, denoted by $\backslash$ and $/$. French strings (of accents) can be given a geometric interpretation as diagrams, as illustrated in Figure 5 left (middle). French strings are to involutive monoids what (ordinary) strings are to monoids. To make this precise, we need the standard notion of a homomorphism as a structure preserving map.

▸ **Definition 5.** A *homomorphism* from the involutive monoid $(A, \cdot, e, -1)$ to the involutive monoid $(B, \cdot', e', -1')$ is a map h from A to B such that for all a, b, c in A, $h(a \cdot b) = h(a) \cdot' h(b)$, $h(e) = e'$, and $h(a^{-1}) = h(a)^{-1'}$. The homomorphism is an *isomorphism* if there exists a homomorphism that is inverse to it.

[4] Meta-footnote: Our naming is tentative. We are open to any suggestion that clearly distinguishes what we call French letters/strings/terms from ordinary ones. We do however insist on using accents because of their intuitive relationship to the geometric representation of conversions as is standard in rewriting since the 1930s (Church–Rosser, Newman), see Figure 5.

▸ **Proposition 6.** *The involutive monoid on French strings $\widehat{L}$ is the free involutive monoid over L. That is, any map from L into the carrier of some involutive monoid, extends, via the map $\ell \mapsto \grave{\ell}$, uniquely to an involutive monoid homomorphism on $\widehat{L}$.*

This is a well-known fact and moreover easy to show. It is implicit in the old proofs of logicality of rewriting for the special case of (abstract) rewrite relations as noted above (recapitulated in Appendix A) and explicitly proven in e.g. [3, Proposition 2].

We conclude this section by giving some examples of homomorphisms linking up the various involutive monoids presented above. These homomorphisms are auxiliary to and illustrative of (see Figure 5) our subsequent constructions of proof orders.

Figure 5 Mapping French strings via strings of accents into area triples.

▸ **Example 7.** (i) Mapping a French string over L to the natural number pair of grave, acute accents in it, is a homomorphism. In turn, mapping a natural number pair to its sum is also a homomorphism. Their composition maps a French string to its *length*, e.g. $\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell} \mapsto (3,2) \mapsto 5$.

(ii) Mapping a French string over L to an ordinary string over L by forgetting accents, is a homomorphism. In turn, mapping a string over L to the multiset of letters in it is also a homomorphism. Their composition maps a French string to its *multiset*, e.g. $\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell} \mapsto \ell\ell mm\ell \mapsto [\ell,\ell,\ell,m,m]$.

(iii) Mapping a French string over L to the French string of its accents by forgetting the letters is a homomorphism. In turn, mapping the accent $\diagdown$ to the area triple $(1,0,0)$ induces a (unique) homomorphism from French strings over accents to area triples. Their composition maps a French string to its *area*, e.g. $\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell} \mapsto \diagup\diagdown\diagdown\diagup\diagdown \mapsto (3,4,2)$, see Figure 5.

4 Proof orders and confluence

In this section we present two novel proof orders, i.e. well-founded orders on proofs in equational logic, factoring these through their interpretation into the French string of their (accented) labels. They are shown both to be proof orders for decreasing diagrams, yielding alternative proofs showing that a locally decreasing rewrite relation is confluent. Both proof orders are flexible, in a sense to be explained in the next section.

4.1 Proof orders via French strings

A proof order is a well-founded order on conversions, i.e. on proofs in equational logic. Proof orders can be generated by proof rewrite systems as introduced in the context of completion by Bachmair and Dershowitz [2]. The objects of a proof rewrite system are conversions and its rewrite steps allow one to replace a subproof, i.e. a conversion between two terms, occurring in it by another such conversion between the same two terms.[5] The idea is to

[5] As before, we deal here only with the special case where the terms are constants.

stepwise transform proofs into simpler ones, the usual goal being to obtain a valley proof (sometimes called a rewrite proof), i.e. a pair of reductions from the source and target of the original conversion, to a common reduct. Here, we adapt these ideas by factoring through an interpretation into the involutive monoid of French strings, the advantage being that they can easily be dealt with algebraically.

▸ **Definition 8.** The *interpretation* of a conversion for an L-indexed $(\underset{\ell}{\to})_{\ell\in L}$ family of rewrite relations, is the French string over L that is the stepwise juxtaposition of the labels in the conversion, where a label carries a grave (acute) accent in case the corresponding step in the conversion is a forward (backward) step.

▸ **Example 9.** The successive conversions of Example 1 are interpreted as the successive French strings in the following transformation, where we have underlined in each step the substring being replaced:

$$\underline{\acute{m}\grave{k}}\acute{\ell}\grave{m} \Rightarrow_{①} \acute{\ell}\acute{m}\underline{\acute{\ell}\grave{m}} \Rightarrow_{②} \acute{\ell}\underline{\acute{m}\grave{m}}\grave{\ell} \Rightarrow_{③} \underline{\acute{\ell}\grave{\ell}}\grave{m}\acute{m}\grave{\ell} \Rightarrow_{④} \acute{\ell}\grave{m}\underline{\acute{m}\grave{\ell}} \Rightarrow_{⑤} \underline{\acute{\ell}\grave{m}}\acute{k}\acute{\ell} \Rightarrow_{⑥} \grave{m}\acute{k}\acute{\ell}$$

Equipping French strings with a well-founded order or with a terminating (French string) rewrite system, gives rise to a proof order, via this interpretation. Among the well-founded orders on French strings, the monotonic ones are of special interest.

▸ **Definition 10.** A *well-founded* involutive monoid is an involutive monoid endowed with a well-founded order $\gg$ on its carrier. It is *monotonic* if the algebraic operations are so with respect to the order, that is, all French string s, t, p satisfy:

1. if $s \gg t$ then $ps \gg pt$ and $sp \gg tp$.
2. if $s \gg t$ then $s^{-1} \gg t^{-1}$

▸ **Theorem 11.** *Let $\succ$ be a well-founded order on L and let the French strings endowed with $\gg$ be a well-founded involutive monoid. Then if for all labels $\ell, m \in L$ and French strings s, r over L (only over $\varnothing$ if $\gg$ is monotonic, i.e. then $s = r = \varepsilon$): $\gg$ is monotonic):*

$$s\acute{\ell}\grave{m}r \gg s\{\ell\succ\}[\grave{m}]\{\ell, m\succ\}[\acute{\ell}]\{m\succ\}r$$

and $(\underset{\ell}{\to})_{\ell\in L}$ is locally decreasing, then $\underset{L}{\to}$ has the Church–Rosser property.

In the statement of the theorem we have employed the EBNF notations [ ] and { } to express option and arbitrary repetition respectively, and used $\vec{\ell}\succ$ to denote a French letter to which (at least) one letter in the vector $\vec{\ell}$ $\succ$-relates. For instance, $[\grave{m}]$ denotes either ε or $\grave{m}$, and $\{\ell\succ\}$ denotes an arbitrary French string of letters to which ℓ $\succ$-relates.

Proof. It suffices to show that any conversion between two objects a and b that is not yet a valley, can be transformed into another conversion between a and b that is more like a valley w.r.t. some well-founded order. If a conversion is not yet a valley, then it contains some local peak, say with interpretation $\acute{\ell}\grave{m}$. By the assumption that the rewrite relation is locally decreasing, the local peak can be transformed into a conversion having interpretation of shape $\{\ell\succ\}[\grave{m}]\{\ell, m\succ\}[\acute{\ell}]\{m\succ\}$, see Figure 2, right. Using the assumption that $s\acute{\ell}\grave{m}r \gg s\{\ell\succ\}[\grave{m}]\{\ell, m\succ\}[\acute{\ell}]\{m\succ\}r$ and well-foundedness of $\gg$, eventually a conversion without local peaks, i.e. a valley proof, is obtained.
If $\gg$ is monotonic, then the comparison for $s = r = \varepsilon$ extends to arbitrary s,r immediately. ◂

A well-founded order satisfying the (displayed) condition of the theorem is called a well-founded involutive monoid *for decreasing diagrams*. The two well-founded involutive monoids for decreasing diagrams to be presented below are obtained via further homomorphisms of the French strings into well-founded involutive monoids. The first one is not monotonic but has 'small' images, whereas the second one is monotonic but has 'large' images.

4.2 An lpo-based order

In this section we turn the involutive monoid of French strings into a well-founded one by showing that it is isomorphic to a set of terms, that we therefore call French terms, and equipping the latter with a certain lexicographic path order. We then show that the resulting monoid is a well-founded involutive monoid for decreasing diagrams.

The simple observation at the basis of our term representation is that when filling in a locally decreasing diagram the multiset (or area) of the French string of *maximal* labels can never increase; labels in the local peak of a locally decreasing diagrams can only cause *smaller* ones to appear in the joining conversion.[6] Accordingly, we recursively stratify a French string into a term having the (French string of its) maximal labels as its head, with the term being of finite height due to the assumed well-foundedness of the order on the labels.

▸ **Definition 12.** The *French* term signature over L is denoted by $L^\sharp_\succ$ and comprises the French strings over L that have $\succ$-incomparable letters, assigning arity zero to ε and to other strings their length plus one. A *French* term over L is a term over $L^\sharp_\succ$ such that each function symbol s occurring in it is related to its ancestor function symbol, say r, by the *Hoare* order for $\succ$, i.e. for each French letter $\hat{\ell}$ in s, there exists a French letter $\hat{m}$ in r such that $m \succ \ell$.

If $\succ$ is the empty relation then the signature comprises all French strings and terms are flat, i.e. have only ε as proper subterms. If on the other hand $\succ$ is total then the signature comprises strings over a single label and the height of a term is the number of distinct labels.

▸ **Example 13.** Well-foundedly ordering the set L of labels of Example 9 as $m \succ \ell, \kappa$, some examples of French terms over L are

$$\acute{m}\grave{m}(\varepsilon, \grave{\kappa}\acute{\ell}(\varepsilon,\varepsilon,\varepsilon), \varepsilon) \qquad \grave{m}\acute{m}(\acute{\ell}\grave{\ell}(\varepsilon,\varepsilon,\varepsilon), \varepsilon, \grave{\ell}(\varepsilon,\varepsilon)) \qquad \grave{m}(\varepsilon, \acute{\kappa}\acute{\ell}(\varepsilon,\varepsilon,\varepsilon))$$

▸ **Lemma 14.** *The inorder-traversal map* $\flat$ flattening *French terms over L into French strings over L, defined inductively by $\varepsilon^\flat = \varepsilon$ and $(\hat{\ell}_1 \dots \hat{\ell}_n(s_0, \dots, s_n))^\flat = t_0^\flat \hat{\ell}_1 s_1^\flat \dots \hat{\ell}_n s_n^\flat$, is a bijection.*

Proof. Let the *stratification* map[7] from French strings over L to French terms over L be inductively defined by setting $\varepsilon^\sharp = \varepsilon$ and $(s_0\hat{\ell}_1 \dots \hat{\ell}_n s_n)^\sharp = \hat{\ell}_1 \dots \hat{\ell}_n(s_0^\sharp, \dots, s_n^\sharp)$ with $n > 0$ and the $\hat{\ell}_i$ all occurrences of $\succ$-maximal French letters in the string. We claim that flattening $\flat$ and stratification $\sharp$ are each other's inverse.

That $\flat \circ \sharp$ is the identity is shown by induction on the length of French strings.

That $\sharp \circ \flat$ is the identity is shown by induction on French terms, using that all function symbols in the direct subterms of a French term are related in the Hoare order to the head. ◂

The images of flattening and stratification are *small*; linear in the size of their input.

▸ **Example 15.** Flattening the terms given in Example 13 yields the French strings $\acute{m}\grave{\kappa}\acute{\ell}\grave{m}$, $\acute{\ell}\grave{\ell}\grave{m}\acute{m}\grave{\ell}$, and $\grave{m}\acute{\kappa}\acute{\ell}$ and stratifying them with respect to $\succ$ yields the original French terms again. These are the interpretations of the first, fourth, and last conversion in Example 1.

---

[6] Forgetting in a first approximation all non-maximal steps while trying to prove confluence by repeatedly filling in locally decreasing diagrams, the diagrams are simply the (square) ones appearing in a diagrammatic proof of the Lemma of Hindley–Rosen. Only at further approximations the non-maximal labels will play a role analogous to the situation in a diagrammatic proof of Newmans's Lemma.

[7] The idea of the stratification map $\sharp$ is a special case of that of the `maxsplit` method/function found in programming languages such as Java/Python.

By the lemma, the French terms over L are (isomorphic to) the free involutive monoid over L, when endowed with the operations on French strings via the bijection, e.g. $t \cdot u = (t^\flat u^\flat)^\sharp$.

▸ **Definition 16.** Let $\succ$ be the relation on the French term signature $L^\sharp_\succ$ defined by interpreting each function symbol, i.e. French string, in $L^\sharp_\succ$ as the pair consisting of its multiset and the middle component of the area (see Example 7), relating these by the lexicographic product of the multiset-extension $\succ_{mul}$ of $\succ$ and the greater-than relation $>$.

We endow French terms with the order $\succ_{ilpo}$ defined as the iterative lexicographic path order [6, Definition 2] induced by the relation $\succ$, where argument places are ordered *lexicographically* by choosing an arbitrary but fixed total order on them *compatible* with the accents, that is, if $\hat{\ell}_{i+1}$ of the $n+1$-ary function symbol $\hat{\ell}_1 \dots \hat{\ell}_n$ has a grave (acute) accent, then the i^{th} argument place of the symbol comes lexicographically before (after) the $i+1^{st}$.[8] For French strings s, t we define $s \succ_{ilpo} t$ as $s^\sharp \succ_{ilpo} t^\sharp$, lifting the strings to French terms first.

▸ **Example 17.** We have $\grave{\kappa}\acute{\ell} \succ \grave{\ell}$ since the multiset $[\ell, \kappa]$ of the former is $\succ_{mul}$-related to the multiset $[\ell]$ of the latter; we have $\acute{\ell}\grave{\ell} \succ \grave{\ell}\acute{\ell}$ since although both have the same multiset $[\ell, \ell]$, the middle component of the area of the former (1) is greater than that of the latter (0).

There are two possible ways to order the argument places of the ternary function symbol $\grave{\kappa}\acute{\ell}$ but the accents dictate that in either case argument place 1 should come after argument places 0 and 2. Similarly, there are two possible ways to order the argument places of $\acute{\ell}\grave{\ell}$ but now 1 should come before the others.

The sequence of transformation steps of Example 9 yields a $\succ_{ilpo}$-sequence:

$\acute{m}\grave{m}(\varepsilon, \grave{\kappa}\acute{\ell}(\varepsilon,\varepsilon,\varepsilon), \varepsilon)$	$\succ_{ilpo}$ ①	decrease of multiset at position 1
$\acute{m}\grave{m}(\acute{\ell}(\varepsilon,\varepsilon), \acute{\ell}(\varepsilon,\varepsilon), \varepsilon)$	$\succ_{ilpo}$ ②	decrease of multiset at position 1
$\acute{m}\grave{m}(\acute{\ell}(\varepsilon,\varepsilon), \varepsilon, \grave{\ell}(\varepsilon,\varepsilon))$	$\succ_{ilpo}$ ③	decrease of area at the root
$\grave{m}\acute{m}(\acute{\ell}\grave{\ell}(\varepsilon,\varepsilon,\varepsilon), \varepsilon, \grave{\ell}(\varepsilon,\varepsilon))$	$\succ_{ilpo}$ ④	decrease of multiset at position 0
$\grave{m}\acute{m}(\acute{\ell}(\varepsilon,\varepsilon), \varepsilon, \grave{\ell}(\varepsilon,\varepsilon))$	$\succ_{ilpo}$ ⑤	decrease of multiset at the root
$\grave{m}(\acute{\ell}(\varepsilon,\varepsilon), \acute{\kappa}\acute{\ell}(\varepsilon,\varepsilon,\varepsilon))$	$\succ_{ilpo}$ ⑥	decrease of multiset at position 0
$\grave{m}(\varepsilon, \acute{\kappa}\acute{\ell}(\varepsilon,\varepsilon,\varepsilon))$		

Note that in the $\succ_{ilpo}$①-step the subterm at position 0 *increases*, but this is not harmful since it comes lexicographically *after* the subterm at position 1 which *decreases*.

▸ **Lemma 18.** *For all labels ℓ,m in L and all French strings s,r over L:*

- $s\hat{\ell}r \succ_{ilpo} s\{\ell>\}r$; *and*
- $s\acute{\ell}\grave{m}r \succ_{ilpo} s\{\ell>\}[\grave{m}]\{\ell, m>\}[\acute{\ell}]\{m>\}r$

Proof. Both items are proved by induction on the length of sr.

- We distinguish cases on whether or not ℓ is $\succ$-maximal in $s\hat{\ell}r$:
  - (yes) in this case one occurrence in ℓ in the head symbol of the lifted lhs $(s\hat{\ell}r)^\sharp$ is replaced by smaller (w.r.t. $\succ$) labels in the head symbol of the lifted rhs, so the lhs' head symbol $\succ$-relates to the rhs' head symbol, and therefore to all rhs function symbols;
  - (no) then we conclude by the induction hypothesis for the substring/term $\hat{\ell}$ is in.
- We distinguish cases on whether or not ℓ, m are $\succ$-maximal in $s\acute{\ell}\grave{m}r$, where $\succ$-maximal labels are coloured blue and non-$\succ$-maximal labels are coloured red:

[8] The chosen order affects $\succ$, but not its compatibility with decreasing diagrams. Argument places can be totally ordered by successively taking the leftmost argument place as allowed by the accents.

(both are) then the head symbol of the lhs $\succ$-relates to the head symbol of the rhs, because the multiset (or else the area) has become smaller, and we conclude as in the (yes)-case above, see Figure 6, left;

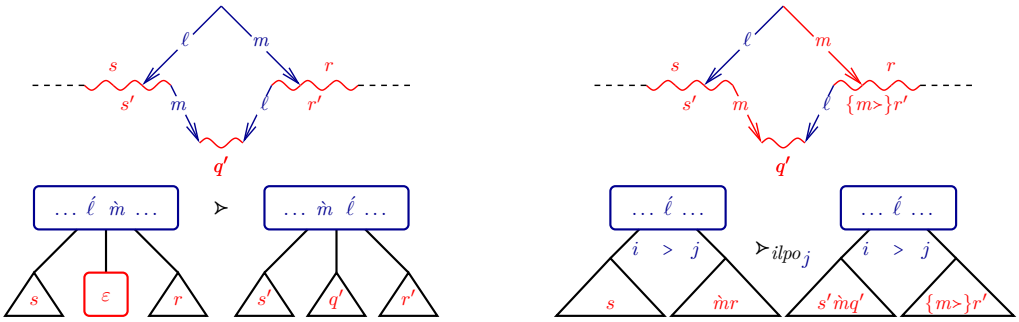


Figure 6 (both,left) Area decrease. (one,right) Lexicographic decrease; jth before ith.

(only $\acute{\ell}$ is) then the substring/term to the right of $\acute{\ell}$ in the lhs $\succ_{ilpo}$-relates to the substring/term to the right of $\acute{\ell}$ in the rhs, using the first item of this lemma, see Figure 6, right;

(only $\grave{m}$ is) as in the previous item but for the substrings/terms to the left of $\grave{m}$;

(neither is) then we conclude by the induction hypothesis for the substring/term the displayed $\acute{\ell}\grave{m}$ is in. ◂

Thus we obtain the main result of this section that French strings endowed with $\succ_{ilpo}$ (via the isomorphism) constitute a well-founded involutive monoid for decreasing diagrams.

Observe that $\succ_{ilpo}$ is *not* monotonic. For instance, composing $\backslash$ to the right of $\backslash/\!/\!/\backslash$ and $/\backslash\backslash\backslash\backslash$ *reverses* the way in which they are ordered by $\succ_{ilpo}$. Moreover, $\succ_{ilpo}$ is not preserved when extending the relation $\succ$. Extending the empty order to $m \succ \ell, \kappa$ *reverses* the way in which $\acute{\ell}\acute{\kappa}\acute{m}$ and $\acute{m}\acute{\kappa}$ are ordered by $\succ_{ilpo}$. Both are overcome by the order introduced in the next section.

▸ Remark. It is possible to overcome non-monotonicity also within the present set-up, the main idea being to quantify over all possible well-orders extending $\succ$ when comparing. More precisely, define the order $\gg$ on the French term signature $L^{\sharp}_{\succ}$ analogous to how $\succ$ was defined in Definition 16, but taking as second component the whole area triple (instead of just its middle component), and comparing these area triples lexicographically on first the pair comprising its first,last component, and then the middle component, with respect to the greater-than (product) order $>$ on the natural numbers. We then define $s \gg r$ to hold if for all well-orders $\succ'$ extending $\succ$ it holds $s \gg'_{ilpo} r$, where $\gg'_{ilpo}$ is the iterative lexicographic path order induced by the order $\gg'$ on $L^{\sharp}_{\succ'}$, which is in turn induced by the extension $\succ'$ of $\succ$. Apart from that the lexicographic order on the argument places should respect the accents as in Definition 16, we now require it to be preserved under concatenation of French strings.

4.3 A monotonic order

▸ **Definition 19.** Let L be an alphabet with precedence $\succ$. We denote by $\gg_{mul}$ and $(\gg_1, \gg_2)_{lex}$ the multiset extension of $\gg$ and the lexicographic product of $\gg_1$ and $\gg_2$, respectively. The order $\gg_\bullet$ on French strings is defined recursively as follows: $s \gg_\bullet t$ iff

$$\langle s \rangle^f \; ((\succ, \gg_\bullet)_{lex})_{mul} \; \langle t \rangle^f$$

where $\langle s\rangle^f = [(\acute{\ell}, q) \mid s = p\acute{\ell}q] \cup [(\grave{\ell}, p) \mid s = p\grave{\ell}q]$ collects acute letters together with their suffix in s and grave letters together with their prefix in s into a multiset, and $\succ$ on French letters just compares their labels. For the following discussion, we define $\gg_\bullet^\Lambda = ((\succ, \gg_\bullet)_{lex})_{mul}$.

Note that Definition 19 is a proper recursive definition: The multiset extension of the lexicographic product of two orders can be computed by comparing only elements present in the compared multisets, and all French strings occurring in $\langle s\rangle^f$ are proper substrings of s.

▸ **Example 20.** Recall the interpretations from Example 9. We show how to compare the first to the last one using the same precedence as in Example 13, i.e. $m \succ \ell, \kappa$. Because $\langle\varepsilon\rangle^f = \varnothing$, while $\langle\grave{\kappa}\acute{\ell}\grave{m}\rangle^f$ is a non-empty multiset, we have $\acute{m}\grave{\kappa}\acute{\ell} \gg_\bullet \varepsilon$. Therefore,

$$\langle\acute{m}\grave{\kappa}\acute{\ell}\grave{m}\rangle^f = [(\acute{m}, \grave{\kappa}\acute{\ell}\grave{m}), (\acute{\ell}, \grave{m}), (\grave{\kappa}, \acute{m}), (\grave{m}, \acute{m}\grave{\kappa}\acute{\ell})] \gg_\bullet^\Lambda [(\acute{\kappa}, \acute{\ell}), (\acute{\ell}, \varepsilon), (\grave{m}, \varepsilon)] = \langle\grave{m}\acute{\kappa}\acute{\ell}\rangle^f,$$
$$\acute{m}\grave{\kappa}\acute{\ell}\grave{m} \gg_\bullet \grave{m}\acute{\kappa}\acute{\ell}.$$

Next we show that $\gg_\bullet$ has all the desired properties: it is a well-founded, monotonic, partial order, provided that $\succ$ is a well-founded order on labels.

▸ **Lemma 21.** *If $\succ$ is a strict partial order on labels, then $\gg_\bullet$ is a strict partial order on French strings. Furthermore the construction is incremental: If $\succ \subseteq \succ'$ then $\gg_\bullet \subseteq \gg'_\bullet$, where $s \gg'_\bullet t$ iff $\langle s\rangle^f\ ((\succ', \gg'_\bullet)_{lex})_{mul}\ \langle t\rangle^f$.*

Proof. Consider the map $\Lambda_\succ(\gg) = \{(s,t) \mid \langle s\rangle^f\ ((\succ, \gg)_{lex})_{mul}\ \langle t\rangle^f\}$. By the properties of the lexicographic product and multiset extension of partial orders, $\Lambda_\succ(\gg)$ is monotonic in $\gg$ (with respect to $\subseteq$) and maps strict partial orders to strict partial orders. Therefore, and because the union of an increasing chain (w.r.t. $\subseteq$) of strict partial orders is again a strict partial order, the least fixed point of $\Lambda_\succ$ exists and is a strict partial order. Inspection of the definition shows that this least fixed point equals $\gg_\bullet$. Incrementality follows because $\Lambda_\succ(\gg)$ is monotonic in $\succ$. ◂

▸ **Lemma 22.** *The order $\gg_\bullet$ on French strings is monotonic.*

Proof. First consider monotonicity of the inverse. We have to show that $s \gg_\bullet t$ implies $s^{-1} \gg_\bullet t^{-1}$. We proceed by induction on the length of s. Note that we can express $\langle s^{-1}\rangle^f$ as $\langle s^{-1}\rangle^f = [(\hat{\ell}^{-1}, p^{-1}) \mid (\hat{\ell}, p) \in \langle s\rangle^f]$. Now by assumption, $\langle s\rangle^f \gg_\bullet^\Lambda \langle t\rangle^f$, and we need to show $\langle s^{-1}\rangle^f \gg_\bullet^\Lambda \langle t^{-1}\rangle^f$, that is,

$$[(\hat{\ell}^{-1}, p^{-1}) \mid (\hat{\ell}, p) \in \langle s\rangle^f] \gg_\bullet^\Lambda [(\hat{\kappa}^{-1}, q^{-1}) \mid (\hat{\kappa}, q) \in \langle t\rangle^f] \tag{1}$$

Since $\hat{\ell}^{-1} \succ \hat{\kappa}^{-1}$ iff $\hat{\ell} \succ \hat{\kappa}$ by definition and $p^{-1} \gg_\bullet q^{-1}$ iff $p \gg_\bullet q$ for all proper substrings p of s by the induction hypothesis, the evaluation of $\langle s\rangle^f \gg_\bullet^\Lambda \langle t\rangle^f$ can be mirrored in the comparison (1), which therefore holds.

Next we show that concatenation is monotonic. Assume that $s \gg_\bullet t$. We need to show that $ps \gg_\bullet pt$ for arbitrary French strings p. (Once we have proved that, we know that $s \gg_\bullet t$ implies $s^{-1} \gg_\bullet t^{-1}$, then $p^{-1}s^{-1} \gg_\bullet p^{-1}t^{-1}$, and finally $sp \gg_\bullet tp$ using the monotonicity of the inverse.) It suffices to show the claim if p has length 1; induction on the length of p will complete the proof. There are two cases, $p = \acute{\ell}$ and $p = \grave{\ell}$. We have:

$$\langle\acute{\ell}s\rangle^f = [(\acute{\kappa}, p) \mid (\acute{\kappa}, p) \in \langle s\rangle^f] \cup [(\grave{\kappa}, \acute{\ell}p) \mid (\grave{\kappa}, p) \in \langle s\rangle^f] \cup [(\acute{\ell}, s)]$$
$$\langle\grave{\ell}s\rangle^f = [(\acute{\kappa}, p) \mid (\acute{\kappa}, p) \in \langle s\rangle^f] \cup [(\grave{\kappa}, \grave{\ell}p) \mid (\grave{\kappa}, p) \in \langle s\rangle^f] \cup [(\grave{\ell}, \varepsilon)]$$

Now when comparing $\langle\acute{\ell}s\rangle^f \gg_\bullet^\Lambda \langle\acute{\ell}t\rangle^f$ (respectively $\langle\grave{\ell}s\rangle^f \gg_\bullet^\Lambda \langle\grave{\ell}t\rangle^f$), we have $(\acute{\ell}, s)\ (\succ, \gg_\bullet)_{lex}$ $(\acute{\ell}, t)$ by assumption ($(\grave{\ell}, \varepsilon) = (\grave{\ell}, \varepsilon)$ trivially), while the other elements of the multisets

originate in $\langle s\rangle^f$ and $\langle t\rangle^f$, and their comparisons carry over to that of $\langle \hat{\ell}s\rangle^f \gg_\bullet \langle \hat{\ell}t\rangle^f$. Note that in the lexicographic product, comparing $(\hat{\kappa}, p)$ and $(\hat{\kappa}', p')$ will only require comparing p to p' if $\hat{\kappa} = \hat{\kappa}'$, and then we know whether $\hat{\ell}$ was prepended to p and p', in which case we apply the induction hypothesis to that comparison, or not. Hence we conclude that $\hat{\ell}s \gg_\bullet \hat{\ell}t$. ◂

▸ Remark. None of the previously mentioned orders are monotonic. We have seen an example for $\succ_{ilpo}$ in Section 4.2; for the order from [5], we have $/\,/\,/\,/\,/ \gg \backslash\,/\,\backslash\,\backslash\,/$ but $/\,\backslash\,/\,\backslash\,\backslash\,/ \gg /\,/\,/\,/\,/\,/$; for [4], $\backslash\,/\,/\,/ \gg \backslash\,/\,\backslash\,/$ but $\backslash\,\backslash\,/\,\backslash\,/ \gg \backslash\,\backslash\,/\,/\,/$.

We still have to establish well-foundedness of $\gg_\bullet$. The proof is based on simple termination [7].

▸ **Theorem 23.** *If the precedence $\succ$ is well-founded, then $\gg_\bullet$ is a well-founded, monotonic, partial order on French strings.*

Proof. By Lemmas 21 and 22, $\gg_\bullet$ is a strict partial order and monotonic. We have to show that $\gg_\bullet$ is well-founded as well. Because its construction is incremental by Lemma 21, we may assume w.l.o.g. that $\succ$ is a (partial) well-order. We can easily see that $\gg_\bullet$ is a simplification order ([7, Definition 5.2]), if we regard French strings a terms over a unary signature as usual: monotonicity means that $\gg_\bullet$ is a rewrite order, while $\hat{\ell} \gg_\bullet \epsilon$ and $\hat{\ell} \gg_\bullet \hat{\kappa}$ if $\hat{\ell} \succ \hat{\kappa}$ ensure that $\succ_{\mathrm{emb}} \subseteq \gg_\bullet$. Therefore, $\gg_\bullet$ is well-founded by [7, Theorem 5.3]. ◂

▸ **Lemma 24.** *Let $\succ$ be a strict partial order. Then (recall the notation from Theorem 11)*

1. *$\hat{\ell} \gg_\bullet \{\ell\succ\}$; and*
2. *$\acute{\ell}\grave{m} \gg_\bullet \{\ell\succ\}[\grave{m}]\{\ell, m\succ\}[\acute{\ell}]\{m\succ\}$*

Proof. 1. Let $t \in \{\ell\succ\}$. We have to establish that $[(\hat{\ell}, \varepsilon)] \gg_\bullet^\Lambda \langle t\rangle^f$. The elements of $\langle t\rangle^f$ are pairs $(\hat{\kappa}, p)$ with $\ell \succ \kappa$, i.e. smaller than $(\hat{\ell}, \varepsilon)$ lexicographically, so the comparison holds.

2. Let $t \in \{\ell\succ\}[\grave{m}]\{\ell, m\succ\}[\acute{\ell}]\{m\succ\}$. We have to establish that $[(\acute{\ell}, \grave{m}), (\grave{m}, \acute{\ell})] \gg_\bullet^\Lambda \langle t\rangle^f$. Most elements of $\langle t\rangle^f$ are pairs $(\hat{\kappa}, p)$ with $\ell \succ \kappa$ or $m \succ \kappa$. There are up to two exceptions, depending on which of the letters $\acute{\ell}$ and $\grave{m}$ are present in t: $(\acute{\ell}, p)$ with $p \in \{m\succ\}$, for which we have $(\acute{\ell}, \grave{m})\ (\succ, \gg_\bullet)_{lex}\ (\acute{\ell}, p)$ using the first case, and $(\grave{m}, p)$ with $p \in \{\ell\succ\}$, for which we have $(\grave{m}, \acute{\ell})\ (\succ, \gg_\bullet)_{lex}\ (\grave{m}, p)$ likewise. Thus, every element of $\langle t\rangle^f$ is dominated by an element of $\langle \acute{\ell}\grave{m}\rangle^f$, and the comparison succeeds. ◂

Consequently, French strings equipped with $\gg_\bullet$ are a well-founded involutive monoid for decreasing diagrams.

We have presented two orders on French strings, $\succ_{ilpo}$ and $\gg_\bullet$. The first order, which we defined by mapping French strings to French terms of size linear in that of the strings, is lightweight and allows an intuitive explanation. The definition of $\gg_\bullet$ is more complex (unfolding it naively will result in an exponential number of comparisons), and opaque. On the other hand, $\succ_{ilpo}$ is not monotonic, which makes the proof of Lemma 18 a bit more tedious than that of Lemma 24. Thanks to monotonicity, the validity of new rules like $\grave{\ell}\grave{m}\grave{m} \Rightarrow \grave{m}\acute{m}\grave{m}\grave{\ell}\acute{m}$ if $\ell \succ m$ is readily established by a direct comparison, $\grave{\ell}\grave{m}\grave{m} \gg_\bullet \grave{m}\acute{m}\grave{m}\grave{\ell}\acute{m}$. Without monotonicity, we would have to consider all possible prefixes and suffixes in the proof.

5 Church–Rosser modulo

In this section we derive a decreasing diagrams technique for Church–Rosser modulo property, in analogy to Section 4. In Section 4.1 we have seen how conversions correspond to French strings. In order to apply this idea to Church–Rosser modulo, we introduce Greek strings, an extension of French strings with self-inverse letters.

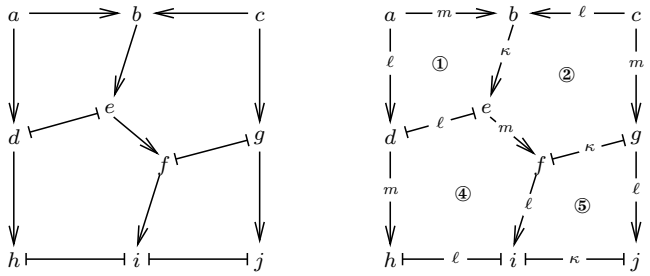


Figure 7 Decomposition of rewrite relations $\rightarrow$, $\mapsto$ enjoying the Church–Rosser modulo property.

5.1 Decreasing diagrams

▸ **Definition 25.** Let L be an alphabet. For each $\ell \in L$ there are three *Greek* letters, accented by acute, grave, or macron accents ($\acute{\ell}$, $\grave{\ell}$, or $\bar{\ell}$). We use $\hat{\ell}$ to denote a Greek letter with label ℓ. Mirroring letters is defined by $\acute{\ell}^{-1} = \grave{\ell}$ and $\bar{\ell}^{-1} = \bar{\ell}$. The *Greek* strings $\overline{L}$ are strings over Greek letters, which together with juxtaposition and mirroring form an involutive monoid. Any precedence $\succ$ on L is extended naturally to Greek letters by letting $\hat{\ell} \succ \hat{m}$ iff $\ell \succ m$. The intended purpose of macron (self-inverse) letters is to represent equational steps in proofs, a natural extension of the interpretations (Definition 8) used for confluence in Section 4.

▸ **Example 26.** Consider the rewrite relations in Figure 7. There are several conversions proving the equivalence of d and g, using labels $L = \{m, \ell, \kappa\}$. We list some interpretations:

$$\underline{\acute{\ell}\grave{m}}\acute{\ell}\grave{m} \Rightarrow_{①} \bar{\ell}\acute{\kappa}\underline{\acute{\ell}\grave{m}} \Rightarrow_{②} \bar{\ell}\underline{\acute{\kappa}\grave{\kappa}}\grave{m}\bar{\kappa} \Rightarrow \underline{\bar{\ell}\grave{m}\bar{\kappa}} \Rightarrow_{④} \grave{m}\bar{\ell}\underline{\acute{\ell}\bar{\kappa}} \Rightarrow_{⑤} \grave{m}\bar{\ell}\bar{\kappa}\acute{\ell}$$

We base our order on the monotonic order from Section 4.3 (Definition 19).

▸ **Definition 27.** Let L be an alphabet with precedence $\succ$. The order $\overline{\gg}_\bullet$ on Greek strings over L is defined by recursion as follows: $s \,\overline{\gg}_\bullet\, t$ iff $\langle s\rangle^g \,((\succ, \overline{\gg}_\bullet)_{lex})_{mul}\, \langle t\rangle^g$ where $\langle s\rangle^g = [(\acute{\ell}, q) \mid s = p\acute{\ell}q] \cup [(\grave{\ell}, p) \mid s = p\grave{\ell}q] \cup [(\bar{\ell}, \varepsilon) \mid s = p\bar{\ell}q]$ collects acute letters together with their suffixes, grave letters together with their prefixes, and macron letters together with empty strings into a multiset. We also define $\overline{\gg}_\bullet^{\Lambda} = ((\succ, \overline{\gg}_\bullet)_{lex})_{mul}$.

▸ Remark. We can regard any French string as a Greek string. If we do that, Definition 27 properly extends Definition 19: The map $\langle\cdot\rangle^g$ is an extension of $\langle\cdot\rangle^f$ that deals with self-inverse letters. One subtle difference is that $\succ$ is also extended: It compares French letters in Definition 19, but Greek letters in Definition 27.

▸ **Example 28.** Continuing Example 26, we show that the second to last step is decreasing, using the order $m \succ \ell \succ \kappa$ on L. In the resulting multiset comparison, it's easy to see that $(\grave{m}, \bar{\ell})$ is larger than every element of the right-hand side multiset:

$$\langle\bar{\ell}\grave{m}\bar{\kappa}\rangle^g = [(\bar{\ell}, \varepsilon), (\grave{m}, \bar{\ell}), (\bar{\kappa}, \varepsilon)] \,\overline{\gg}_\bullet^{\Lambda}\, [(\grave{m}, \varepsilon), (\bar{\ell}, \varepsilon), (\acute{\ell}, \bar{\kappa}), (\bar{\kappa}, \varepsilon)] = \langle\grave{m}\bar{\ell}\acute{\ell}\bar{\kappa}\rangle^g$$
$$\bar{\ell}\grave{m}\bar{\kappa} \,\overline{\gg}_\bullet\, \grave{m}\bar{\ell}\acute{\ell}\bar{\kappa}$$

The order $\overline{\gg}_\bullet$ shares many properties with $\gg_\bullet$.

▸ **Theorem 29.** *If the precedence $\succ$ on L is well-founded, then the order $\overline{\gg}_\bullet$ is a well-founded, monotonic, partial order on Greek strings.*

Proof. The similarities between $\gg_\bullet$ and $\overline{\gg}_\bullet$ are so overwhelming that the proofs of Lemmas 21, 22 and Theorem 23 work with straight-forward modifications:

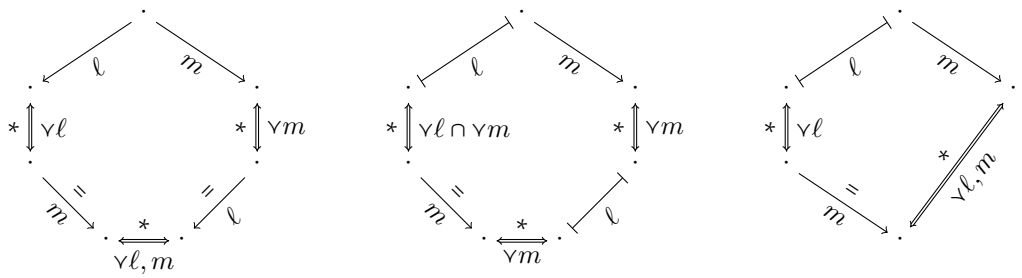


Figure 8 Locally decreasing diagrams for Church–Rosser modulo.

- Replace $\gg_\bullet$ by $\overline{\gg}_\bullet$, $\gg_\bullet^\Lambda$ by $\overline{\gg}_\bullet^\Lambda$ and $\langle\cdot\rangle^f$ by $\langle\cdot\rangle^g$ everywhere.
- In Lemma 21, define Λ by $\Lambda(\gg) = \{(s,t) \mid \langle s\rangle^g \; ((>, \gg)_{lex})_{mul} \; \langle t\rangle^g\}$.
- In Lemma 22, the expression for $\langle s^{-1}\rangle^f$ remains valid for $\langle s^{-1}\rangle^g$. For the monotonicity of concatenation, we have to consider three cases for p of length 1, $p = \acute{\ell}$, $p = \grave{\ell}$ and $p = \bar{\ell}$, and we can express $\langle \hat{\ell} s\rangle^g$ as follows:

$$\langle \acute{\ell} s\rangle^g = [(\hat{\kappa}, p) \mid (\hat{\kappa}, p) \in \langle s\rangle^g \text{ and } \hat{\kappa} \neq \grave{\kappa}] \cup [(\grave{\kappa}, \acute{\ell} p) \mid (\grave{\kappa}, p) \in \langle s\rangle^g] \cup [(\acute{\ell}, s)]$$
$$\langle \grave{\ell} s\rangle^g = [(\hat{\kappa}, p) \mid (\hat{\kappa}, p) \in \langle s\rangle^g \text{ and } \hat{\kappa} \neq \grave{\kappa}] \cup [(\grave{\kappa}, \grave{\ell} p) \mid (\grave{\kappa}, p) \in \langle s\rangle^g] \cup [(\grave{\ell}, \varepsilon)]$$
$$\langle \bar{\ell} s\rangle^g = [(\hat{\kappa}, p) \mid (\hat{\kappa}, p) \in \langle s\rangle^g \text{ and } \hat{\kappa} \neq \grave{\kappa}] \cup [(\grave{\kappa}, \bar{\ell} p) \mid (\grave{\kappa}, p) \in \langle s\rangle^g] \cup [(\bar{\ell}, \varepsilon)]$$

  When comparing $\langle \bar{\ell} s\rangle^g$ and $\langle \bar{\ell} t\rangle^g$, we have $(\bar{\ell}, \varepsilon) = (\bar{\ell}, \varepsilon)$, and the remaining elements of the multisets originate in $\langle s\rangle^g$ and $\langle t\rangle^g$, respectively. The remainder of the argument in Lemma 22 applies directly.
- Finally, the well-foundedness proof in Theorem 23 requires no further modifications. ◂

▸ **Lemma 30.** *Let $\succ$ be a strict partial order. Then (recall the notation from Theorem 11)*

1. $\hat{\ell} \; \overline{\gg}_\bullet \; \{\ell>\}$ *and* $\acute{\ell}\grave{m} \; \overline{\gg}_\bullet \; \{\ell>\}[\grave{m}]\{\ell, m>\}[\acute{\ell}]\{m>\}$*;*
2. $\bar{\ell}\grave{m} \; \overline{\gg}_\bullet \; (\{\ell>\} \cap \{m>\})[\grave{m}]\{m>\}\bar{\ell}\{m>\}$ *(the intersection works on sets of strings); and*
3. $\bar{\ell}\grave{m} \; \overline{\gg}_\bullet \; \{\ell>\}[\grave{m}]\{\ell, m>\}$*.*

Proof. 1. The first item is analogous to Lemma 24.

2. Let $t \in (\{\ell>\} \cap \{m>\})[\grave{m}]\{m>\}\bar{\ell}\{m>\}$. We have to show that $[(\bar{\ell}, \varepsilon), (\grave{m}, \bar{\ell})] \; \overline{\gg}_\bullet^\Lambda \; \langle t\rangle^g$. Note that $(\bar{\ell}, \varepsilon) \in \langle t\rangle^g$, so all other elements of $\langle t\rangle^g$ must be smaller than $(\grave{m}, \bar{\ell})$. This is true for $(\grave{m}, p)$ with $p \in (\{\ell>\} \cap \{m>\})$ by the first item of this lemma, and all remaining pairs $(\hat{\kappa}, p) \in \langle t\rangle^g$ are smaller than $(\grave{m}, \bar{\ell})$ because $m \succ \kappa$. We conclude that $\bar{\ell}\grave{m} \; \overline{\gg}_\bullet \; t$.
3. Let $t \in \{\ell>\}[\grave{m}]\{\ell, m>\}$. We show that $[(\bar{\ell}, \varepsilon), (\grave{m}, \bar{\ell})] \; \overline{\gg}_\bullet^\Lambda \; \langle t\rangle^g$. All elements $(\hat{\kappa}, p)$ of $\langle t\rangle^g$ have $\ell \succ \kappa$ or $r \succ \kappa$ (and are thus smaller than one of $(\grave{m}, \bar{\ell})$ or $(\bar{\ell}, \varepsilon)$), with one possible exception: $(\grave{m}, p)$ where $p \in \{\ell>\}$, which is smaller than $(\grave{m}, \bar{\ell})$ using the first item of this lemma. Therefore, $\bar{\ell}\grave{m} \; \overline{\gg}_\bullet \; t$ is true. ◂

▸ **Theorem 31.** *Let L be an alphabet equipped with a well-founded order $\succ$. Furthermore, let $(\underset{\ell}{\to})_{\ell \in L}$ and $(\underset{\ell}{\mapsto})_{\ell \in L}$ be families of abstract rewrite relations, where each $\underset{\ell}{\mapsto}$ is symmetric. If*

$$\underset{\ell}{\leftarrow} \cdot \underset{m}{\longrightarrow} \subseteq \overset{*}{\underset{\gamma\ell}{\Leftrightarrow}} \cdot \underset{m}{\overset{=}{\longrightarrow}} \cdot \overset{*}{\underset{\gamma\ell, m}{\Longleftrightarrow}} \cdot \underset{\ell}{\overset{=}{\leftarrow}} \cdot \overset{*}{\underset{\gamma m}{\Longleftrightarrow}}$$

and

$$\underset{\ell}{\mapsto} \cdot \underset{m}{\longrightarrow} \subseteq \left(\overset{*}{\underset{\gamma\ell \cap \gamma m}{\Longleftrightarrow}} \cdot \underset{m}{\overset{=}{\longrightarrow}} \cdot \overset{*}{\underset{\gamma m}{\Longleftrightarrow}} \cdot \underset{\ell}{\mapsto} \cdot \overset{*}{\underset{\gamma m}{\Longleftrightarrow}} \right) \cup \left(\overset{*}{\underset{\gamma\ell}{\Leftrightarrow}} \cdot \underset{m}{\overset{=}{\longrightarrow}} \cdot \overset{*}{\underset{\gamma\ell, m}{\Longleftrightarrow}} \right),$$

for all $\ell, m \in L$, where $\underset{\ell}{\Leftrightarrow} = \underset{\ell}{\leftarrow} \cup \underset{\ell}{\mapsto} \cup \underset{\ell}{\to}$ (see Figure 8), then $\underset{L}{\to}$ is Church–Rosser modulo $\underset{L}{\mapsto}$.

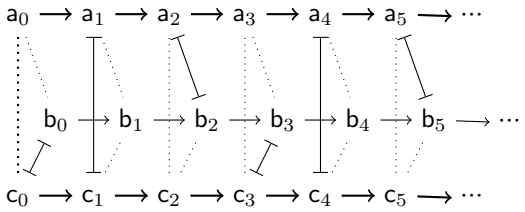


Figure 9 Incompleteness: The rewrite relations $\to$ and $\mapsto$.

Proof. The proof follows that of Theorem 11. First we observe that if a conversion between two objects a and b is not a valley of shape $\overset{*}{\to} \cdot \overset{*}{\mapsto} \cdot \overset{*}{\leftarrow}$, then it must contain a local peak or cliff. By assumption, we can replace that peak or cliff by an alternative subproof. To show termination, we observe that the interpretation of the replacement proof is smaller than that of the peak or cliff w.r.t. $\overline{\gg_\bullet}$, by Lemma 30. Thanks to monotonicity this extends to the interpretations of the whole proofs. This implies termination, because $\overline{\gg_\bullet}$ is well-founded. ◂

The rewrite relations in Figure 7 have the Church–Rosser modulo property, because every local peak and cliff can be joined in a decreasing diagram of the required shape. As an instance of Theorem 31 we obtain the following result by Jouannaud and Liu.

▸ **Corollary 32** ([4, Corollary 2.5.8]). *Let $(\underset{\ell}{\to})_{\ell\in L}$ and $(\underset{\ell}{\mapsto})_{\ell\in L}$ be families of abstract rewrite relations, where each $\underset{\ell}{\mapsto}$ is symmetric. Then $\underset{L}{\to}$ is Church–Rosser modulo $\underset{L}{\mapsto}$, if for all $\ell, m \in L$, $\underset{\ell}{\leftarrow} \cdot \underset{m}{\to} \subseteq \overset{*}{\underset{\curlyvee\ell}{\Leftrightarrow}} \cdot \overset{=}{\underset{m}{\to}} \cdot \overset{*}{\underset{\curlyvee\ell,m}{\Leftrightarrow}} \cdot \overset{=}{\underset{\ell}{\leftarrow}} \cdot \overset{*}{\underset{\curlyvee m}{\Leftrightarrow}}$ and $\underset{\ell}{\mapsto} \cdot \underset{m}{\to} \subseteq \overset{=}{\underset{m}{\to}} \cdot \overset{*}{\underset{\curlyvee\ell,m}{\Leftrightarrow}}$.*

Furthermore, Ohlebusch's Main Theorem of [8] is a consequence of Corollary 32 by labelling all $\mapsto$ steps with a minimal, fresh label $\bot$. As another instance of Theorem 31 we can obtain a key lemma for abstract Church–Rosser modulo from [1]:

▸ **Corollary 33** (Aoto and Toyama [1, Lemma 2.1]). *Let $(\underset{\ell}{\to})_{\ell\in L}$ and $(\underset{\ell}{\mapsto})_{\ell\in L}$ be families of abstract rewrite relations, where each $\underset{\ell}{\mapsto}$ is symmetric. Then $\underset{L}{\to}$ is Church–Rosser modulo $\underset{L}{\mapsto}$, if for all $\ell, m \in L$, $\underset{\ell}{\leftarrow} \cdot \underset{m}{\to} \subseteq \underset{\curlyvee\ell,m}{\Leftrightarrow}$ and $\underset{\ell}{\mapsto} \cdot \underset{m}{\to} \subseteq \underset{\curlyvee\ell,m}{\Leftrightarrow}$.*

5.2 Incompleteness

It is known that decreasing diagrams are complete for confluence of countable rewrite relations [9, Theorem 14.2.32]. In this section we show that no terminating proof rewrite system can be complete for proving Church–Rosser modulo. To this end, we exhibit a pair of rewrite relations $\to$, $\mapsto$ such that $\to$ is Church–Rosser modulo $\mapsto$, but there is no terminating proof rewrite system that has only valley proofs of shape $\overset{*}{\to} \cdot \overset{*}{\mapsto} \cdot \overset{*}{\leftarrow}$ as normal forms.

▸ Remark. Terminating proof rewrite systems are exactly those which are compatible with some monotonic well-founded order on proofs. However, as we have seen with $\succ_{ilpo}$ in Section 4.2, we can also show termination of proof rewrite systems using non-monotonic orders. The incompleteness result of this section applies to such proofs as well.

▸ **Definition 34.** On the set $A = \{\mathsf{a}_i, \mathsf{b}_i, \mathsf{c}_i \mid i \in \mathbb{N}\}$ we define the relations $\to$ and $\mapsto$ as follows:

1. $u_i \to u_{i+1}$ iff $u \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$;

2. $u_i \mapsto v_i$ iff $\{u,v\} = \{\mathsf{b},\mathsf{c}\}$ if $i \equiv 0 \pmod 3$, $\{u,v\} = \{\mathsf{a},\mathsf{c}\}$ if $i \equiv 1 \pmod 3$ and $\{u,v\} = \{\mathsf{a},\mathsf{b}\}$ otherwise. (See also Figure 9.)

▸ Remark. Definition 34 may be regarded as a simplified version of [4, Figure 1(a)]. Both examples would serve the purpose of this section, and Theorem 36 subsumes the incompleteness result of [4, Section 4.3].

Note that $\to$ is deterministic and that all $\overset{*}{\mapsto}$ equivalence classes have size 1 or 2. Together with the periodic and symmetric nature of the rewrite relations (consider mapping a_i, b_i and c_i to b_{i+1}, c_{i+1} and a_{i+1}, respectively), this restricts valley proofs to just a few possibilities:

▸ **Proposition 35.** *1. The rewrite relation $\to$ is Church–Rosser modulo $\mapsto$.*
2. Any valley proof for a peak $\overset{n}{\leftarrow} \cdot \overset{m}{\to}$ has shape $\overset{l-n}{\longrightarrow} \cdot \overset{*}{\mapsto} \cdot \overset{l-m}{\longleftarrow}$ for some $l \geqslant n, m \geqslant 0$.
3. Any valley proof for a local cliff $\leftarrow \cdot \mapsto$ has shape $\overset{3n-1}{\longrightarrow} \cdot \overset{+}{\mapsto} \cdot \overset{3n}{\longleftarrow}$ for some $n > 0$.

The following result establishes that no terminating proof rewrite system can be complete for Church–Rosser modulo of $\to$, $\mapsto$.

▸ **Theorem 36.** *There is no terminating proof rewrite system for $\to$, $\mapsto$ that only rewrites local peaks and cliffs and always produces valley proofs as normal forms.*

Proof. By contradiction. Assume that we are given a terminating proof rewrite system whose normal forms are valley proofs. We show coinductively that any proof of shape

$$\mapsto \cdot \overset{n}{\to} \cdot \overset{m}{\leftarrow} \cdot \mapsto \tag{2}$$

with $n \not\equiv m \pmod 3$ allows an infinite proof rewrite sequence. Note that such proofs exist, for example, we have $\mathsf{b}_0 \mapsto \mathsf{c}_0 \to \mathsf{c}_1 \mapsto \mathsf{a}_1$. We may assume w.l.o.g. that $n > 0$ (if $n = 0$, then $m > 0$, and we can conclude symmetrically). Then we can rewrite the initial cliff $\mapsto \cdot \to$ to a normal form, which must be a valley proof. By Proposition 35, the resulting proof has shape $\overset{3k}{\longrightarrow} \cdot \overset{+}{\mapsto} \cdot \overset{3k-1}{\longleftarrow} \cdot \overset{n-1}{\longrightarrow} \cdot \overset{m}{\leftarrow} \cdot \mapsto$ for some $k \in \mathbb{N}$. Similarly, we can reduce the new peak $\overset{3k-1}{\longleftarrow} \cdot \overset{n-1}{\longrightarrow}$ to a valley proof, which by Proposition 35 results in a proof

$$\overset{3k}{\longrightarrow} \cdot \overset{+}{\mapsto} \cdot \overset{u}{\to} \cdot \overset{p}{\mapsto} \cdot \overset{v}{\leftarrow} \cdot \mapsto \tag{3}$$

with $u = l - 3k + 1$ and $v = l - n + 1 + m$ for some $l, p \in \mathbb{N}$. for some $l, p \in \mathbb{N}$. Let $u = l - 3k + 1$ and $v = l - n + 1 + m$. It is easy to see that $u \not\equiv v \pmod 3$. If $p = 0$, then (3) contains a subproof of shape (2), namely $\mapsto \cdot \overset{u}{\to} \cdot \overset{v}{\leftarrow} \cdot \mapsto$. If $p > 0$ and $u \not\equiv 0 \pmod 3$ then the subproof $\mapsto \cdot \overset{u}{\to} \cdot \overset{0}{\leftarrow} \cdot \mapsto$ of (3) has shape (2). Otherwise, $p > 0$ and $v \not\equiv 0 \pmod 3$, and the subproof $\mapsto \cdot \overset{0}{\to} \cdot \overset{v}{\leftarrow} \cdot \mapsto$ of (3) has shape (2). Continuing this process on the obtained subproof, we obtain an infinite proof rewrite sequence, contradicting our termination assumption. ◂

▸ Remark. Note that by identifying u_i with u_{i+3} for all $i \in \mathbb{N}$ and $u \in \{\mathsf{a},\mathsf{b},\mathsf{c}\}$ we obtain a pair of finite rewrite relations for which Theorem 36 still holds.

6 Conclusion

We have presented two well-founded orders on French strings that entail the decreasing diagrams technique, one based on (i)lpo and the other with a more complex definition (related to rpo) that makes it monotone. Generalising the monotone order to work on Greek strings that include self-inverse letters, we have obtained a new result for Church–Rosser modulo. It would be interesting to generalise the ilpo based order as well, but we leave that to future work. Finally, we have shown that no complete criterion for Church–Rosser modulo can be

obtained by considering proof transformations alone; at the least, some sort of strategy for applying proof rewrite rules must be incorporated.

Acknowledgements The first author is supported by FWF (Austrian Science Fund) project P22467. The question of completeness (Section 5.2) was raised by Jouannaud. We would also like to thank the anonymous reviewers for their valuable feedback. In particular, the short proof of Theorem 29 was suggested by one of the reviewers.

A Appendix

Proof of Proposition 6. Consider the term rewrite system obtained by orienting the laws of Definition 2 from left to right into term rewrite rules:

$$\begin{array}{rcl@{\qquad}rcl} c(c(x,y),z) & \to & c(x,c(y,z)) & i(i(x)) & \to & x \\ c(x,e) & \to & x & i(c(x,y)) & \to & c(i(y),i(x)) \\ c(e,x) & \to & x & i(e) & \to & e \end{array}$$

This term rewriting system is confluent and terminating, as tools nowadays can show automatically, and has as closed normal forms[9] e and the elements of the set N given by:

$$N ::= \ell \mid i(\ell) \mid c(\ell, N) \mid c(i(\ell), N)$$

Therefore, endowing $\{e\} \cup N$ with operations c, e, and i, in each case followed by taking normal forms, constitutes a free involutive monoid. This monoid is easily seen to be isomorphic to the one on French strings via the bijection between N and $\widehat{L}$ induced by $\ell \mapsto \grave{\ell}$. ◂

References

1 T. Aoto and Y. Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *LMCS*, 8(1:31):1–29, 2012.
2 L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *Journal of the ACM*, 41(2):236–276, 1994.
3 B. Jacobs. Involutive categories and monoids, with a GNS-correspondence. *Foundations of Physics*, pages 1–22, 2011.
4 J. P. Jouannaud and Jiaxiang Liu. From diagrammatic confluence to modularity. *Theoretical Computer Science*, 464:20–34, 2012.
5 J.-P. Jouannaud and V. van Oostrom. Diagrammatic confluence and completion. In *Proc. 36th ICALP*, volume 5556 of *LNCS*, pages 212–222, 2009.
6 J.W. Klop, V. van Oostrom, and R. de Vrijer. Iterative lexicographic path orders. In *Algebra, Meaning and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *LNCS*, pages 541–554, 2006.
7 A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theoretical Computer Science*, 175(1):127–158, 1997.
8 E. Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In *Proc. 9th RTA*, volume 1379 of *LNCS*, pages 17–31, 1998.
9 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
10 V. van Oostrom. Confluence by decreasing diagrams – converted. In *Proc. 19th RTA*, volume 5117 of *LNCS*, pages 306–320, 2008.

[9] Think of these closed normal forms as (empty) conversions.

Decidable structures between Church-style and Curry-style

Ken-etsu Fujita[1] **and Aleksy Schubert**[2]

1 **Gunma University**
Tenjin-cho 1-5-1, Kiryu 376-8515, Japan
fujita@cs.gunma-u.ac.jp

2 **The University of Warsaw**
ul. Banacha 2, 02-097 Warsaw, Poland
alx@mimuw.edu.pl

Abstract

It is well-known that the type-checking and type-inference problems are undecidable for second order λ-calculus in Curry-style, although those for Church-style are decidable. What causes the differences in decidability and undecidability on the problems? We examine crucial conditions on terms for the (un)decidability property from the viewpoint of partially typed terms, and what kinds of type annotations are essential for (un)decidability of type-related problems. It is revealed that there exists an intermediate structure of second order λ-terms, called a style of hole-application, between Church-style and Curry-style, such that the type-related problems are decidable under the structure. We also extend this idea to the omega-order polymorphic calculus F_ω, and show that the type-checking and type-inference problems then become undecidable.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.4.1 Mathematical Logic

Keywords and phrases 2nd-order λ-calculus, type-checking, type-inference, Church-style and Curry-style

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.190

1 Introduction

Traditionally, following the fathers [6, 7], we have two styles of λ-terms with types [2], Church-style[1] and Curry-style. Terms in the style of Church contain full type annotation, so that this style enjoys uniqueness of typing derivations. On the other hand, terms in the style of Curry are the same as those of the type free λ-calculus, and a type inference algorithm may compute their types.

The two styles give no distinction to solvability of type-related problems of simply typed λ-calculus. However, in the case of second order λ-calculus (Girard and Reynolds), it is well-known that the type checking and type inference problems are decidable for Church-style[2] but undecidable for Curry-style [32]. The two definitions of λ-terms are so different, and our motivation behind this work is to make it clear what is a crucial condition on terms for the (un)decidable property of the problems.

[1] The terminology, Church-style terms here are also called pseudo-terms *á la* de Bruijn in the recent literature [3].

[2] The problems are, in general, decidable for normalizing PTS (Pure Type Systems) with a finite set of sorts [31].

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 190–205

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Decidability of TCP, TIP, and TPP for $\lambda 2$-terms with intermediate styles.

Styles	TCP		TIP		TPP
Church	Yes	$\hookrightarrow$	Yes	$\hookrightarrow$	No [30]
Hole-application	Yes	$\longleftrightarrow$	Yes	$\hookrightarrow$	No
Domain-free	No [11]	$\longleftrightarrow$	No [11]	$\longleftrightarrow$	No [24]
Type-free	No [13]	$\longleftrightarrow$	No [13]	$\hookleftarrow$	No [13]
Curry	No [32]	$\longleftrightarrow$	No [32]	$\hookleftarrow$	No [32]

For this principal objective, we introduce three intermediate structures called domain-free, hole-application, and type-free, see Table 1, between Church-style and Curry-style based on the previous work [11]. In the table, TCP denotes type checking, TIP type inference, and TPP typability problems, respectively. "Yes" means that a corresponding problem is decidable, and "No" undecidable. Arrows ($\hookrightarrow, \longleftrightarrow, \hookleftarrow$) denote reduction relations between problems following forthcoming Proposition 4. Our idea on the intermediate structures is quite natural from the viewpoints of type erasure mapping and partially typed terms. Terms in the style of hole-application contain domains of λ-abstraction $\lambda x\!:\!A.M$ just like Church-style, but omit the information on a polymorphic instance such as $M[]$ instead of $M[A]$. On the other hand, terms in the style of domain-free contain the information on a polymorphic instance $M[A]$ like Church-style, but omit domains of λ-abstraction such as $\lambda x.M$ rather than $\lambda x\!:\!A.M$. Terms in the style of type-free contain no type information at all like Curry-style, but contain information holders [] to be filled with a type. We will introduce an order on the styles via type erasure mappings, and in terms of the intermediate structures, we will identify the boundaries between decidability and undecidability with respect to the type-related problems, see also Figure 1 in the next section.

The partial type reconstruction problem can be regarded as a type inference problem for mixed styles of the intermediate structures. Following Boehm [5] and Pfenning [26, 27], partial type reconstruction is in general undecidable. From the viewpoint of partially typed terms, the intermediate structures including Church and Curry-styles can be regarded as a unifying framework, under which various systems can be compared comfortably.

Our work concerns both theoretical and practical aspects of programming. From the perspective of designing programming languages, we investigate a trade-off between decidability for type-related problems and comfortable programming with less annotations (overheads) in terms. This paper makes the following particularly theoretical contributions (i, ii, iii).

It is proved that (i) TIP is decidable (Yes in Table 1), but (ii) TPP is undecidable (No in Table 1) for hole-application $\lambda 2$. Hence, compared with Church-style, type inference problems remain decidable even after deleting polymorphic instance information $M[]$ from $M[A]$, but deleting a polymorphic domain $\lambda x.M$ from $\lambda x\!:\!A.M$ makes the problems undecidable. The introduction of hole-application reveals that putting polymorphic domains on terms is very important to design systems with decidable type inference. Notably, the annotation of function signatures with types is used in main-stream languages such as C or Java, so this annotational overhead seems to be acceptable for the community of programmers.

Finally, we extend this idea to the omega-order polymorphic calculus F_ω, and then show that (iii) TCP and TIP for hole-application F_ω become undecidable.

This paper is organized as follows. We introduce the second order λ-calculus $\lambda 2$ in five styles and basic definitions, and show fundamental properties of the system in Section 2. Section 3 demonstrates that the typability problem for hole-application $\lambda 2$ is undecidable. Next, a type inference algorithm for hole-application $\lambda 2$ is provided, and we prove that the

algorithm is sound and complete. Then, the subject reduction property for hole-application $\lambda 2$ is proved. Section 4 handles hole-application F_ω. Section 5 summarizes results for $\lambda 2$ in five styles (Table 1), concluding remarks, and related work.

2 Second-order lambda-calculus $\lambda 2$ in five styles

2.1 Church-style and Curry-style $\lambda 2$

We introduce the second-order lambda-calculus $\lambda 2$ (Girard and Reynolds) in the styles of Church and Curry, respectively. Types, λ-terms for each style, and inference rules are usually defined as follows:

▶ **Definition 1** ($\lambda 2$ in Church-style and Curry-style).

- $\lambda 2$-types:
$$A ::= X \mid (A \to A) \mid \forall X.A$$
- $\lambda 2$-terms in Church-style:
$$M ::= x \mid (\lambda x{:}A.M) \mid (MM) \mid (\Lambda X.M) \mid (M[A])$$
- Contexts:
A context denoted by Γ or Σ is a set of a declaration of the form $x : A$ with distinct variables as subjects. We write $\Gamma(x) = A$ for $x{:}A \in \Gamma$ and $\mathrm{dom}(\Gamma)$ for $\{x \mid x{:}A \in \Gamma\}$.
- Inference rules for Church-style:
$$\frac{}{\Gamma, x{:}A \vdash_{\mathrm{Ch}} x : A}\ (\mathrm{var})$$
$$\frac{\Gamma, x{:}A_1 \vdash_{\mathrm{Ch}} M : A_2}{\Gamma \vdash_{\mathrm{Ch}} \lambda x{:}A_1.M : A_1 \to A_2}\ (\to I) \qquad \frac{\Gamma \vdash_{\mathrm{Ch}} M_1 : A_1 \to A_2 \quad \Gamma \vdash_{\mathrm{Ch}} M_2 : A_1}{\Gamma \vdash_{\mathrm{Ch}} M_1 M_2 : A_2}\ (\to E)$$
$$\frac{\Gamma \vdash_{\mathrm{Ch}} M : A}{\Gamma \vdash_{\mathrm{Ch}} \Lambda X.M : \forall X.A}\ (\forall I)^\star \qquad \frac{\Gamma \vdash_{\mathrm{Ch}} M : \forall X.A}{\Gamma \vdash_{\mathrm{Ch}} M[A_1] : A[X := A_1]}\ (\forall E)$$
where $(\forall I)^\star$ denotes that the eigenvariable condition $X \notin \mathrm{FV}(\Gamma)$ is imposed on the application, such that X never appears free in Γ.
- $\lambda 2$-terms in Curry-style:
$$M ::= x \mid (\lambda x.M) \mid (MM)$$
Inference rules for Curry-style:
$$\frac{}{\Gamma, x{:}A \vdash_{\mathrm{Cu}} x : A}\ (\mathrm{var})$$
$$\frac{\Gamma, x{:}A_1 \vdash_{\mathrm{Cu}} M : A_2}{\Gamma \vdash_{\mathrm{Cu}} \lambda x.M : A_1 \to A_2}\ (\to I) \qquad \frac{\Gamma \vdash_{\mathrm{Cu}} M_1 : A_1 \to A_2 \quad \Gamma \vdash_{\mathrm{Cu}} M_2 : A_1}{\Gamma \vdash_{\mathrm{Cu}} M_1 M_2 : A_2}\ (\to E)$$
$$\frac{\Gamma \vdash_{\mathrm{Cu}} M : A}{\Gamma \vdash_{\mathrm{Cu}} M : \forall X.A}\ (\forall I)^\star \qquad \frac{\Gamma \vdash_{\mathrm{Cu}} M : \forall X.A}{\Gamma \vdash_{\mathrm{Cu}} M : A[X := A_1]}\ (\forall E)$$
where $(\forall I)^\star$ denotes the eigenvariable condition $X \notin \mathrm{FV}(\Gamma)$.

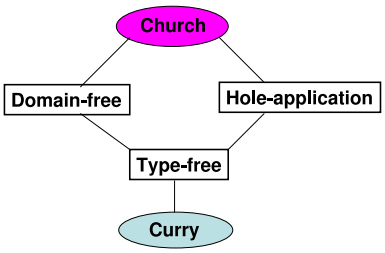


Figure 1 $\lambda 2$-terms with intermediate structures between Curry-style and Church-style.

2.2 Intermediate structures between Church and Curry

Next we define λ-terms with intermediate structures [11] between Church-style and Curry-style, which are called domain-free, hole-application, and type-free. We simply write Ch, Df, Ha, Tf, and Cu, respectively, for the styles, and we employ the terminology λ-terms in s-style for each $s \in \{\mathrm{Ch}, \mathrm{Df}, \mathrm{Ha}, \mathrm{Tf}, \mathrm{Ch}\}$.

▶ **Definition 2** (Domain-free, hole-application, and type-free).

- Domain-free style $\lambda 2$-terms:

$$M ::= x \mid (\lambda x.M) \mid (MM) \mid (\Lambda X.M) \mid (M[A])$$

- Hole-application style $\lambda 2$-terms:

$$M ::= x \mid (\lambda x\!:\!A.M) \mid (MM) \mid (\Lambda X.M) \mid (M[])$$

- Type-free style $\lambda 2$-terms:

$$M ::= x \mid (\lambda x.M) \mid (MM) \mid (\Lambda.M) \mid (M[])$$

  Inference rules for domain-free, hole-application, and type-free styles, respectively, are defined similarly.

Based on the Curry-Howard isomorphism [17], well-typed $\lambda 2$-terms play the role of codes for proofs. From the viewpoint of proof codes, well-typed λ-terms contain three kinds of information on proofs: (i) what inference rule is applied, (ii) where it is applied, and (iii) how to instantiate a rule. A mapping, which erases some of above information one by one from Church-style provides more abstract λ-terms with an intermediate structure. We examine what kind of information on λ-terms is the most essential for (un)decidability of type-related problems.

▶ **Definition 3** (Order on styles and erasure mapping). We define an order on the styles, see Figure 1, such that $\mathrm{Cu} < \mathrm{Tf} < \mathrm{Df} < \mathrm{Ch}$ and $\mathrm{Tf} < \mathrm{Ha} < \mathrm{Ch}$. For styles $s, t \in \{\mathrm{Cu}, \mathrm{Tf}, \mathrm{Ha}, \mathrm{Df}, \mathrm{Ch}\}$ with $s < t$, an erasure $|\cdot|_s^t$ is defined naturally as a function from t-style $\lambda 2$-terms to s-style $\lambda 2$-terms as follows:

- $|x|_{\mathrm{Df}}^{\mathrm{Ch}} = x$, $|\lambda x\!:\!A.M|_{\mathrm{Df}}^{\mathrm{Ch}} = \lambda x.|M|_{\mathrm{Df}}^{\mathrm{Ch}}$, $|M_1 M_2|_{\mathrm{Df}}^{\mathrm{Ch}} = |M_1|_{\mathrm{Df}}^{\mathrm{Ch}}|M_2|_{\mathrm{Df}}^{\mathrm{Ch}}$,
  $|\Lambda X.M|_{\mathrm{Df}}^{\mathrm{Ch}} = \Lambda X.|M|_{\mathrm{Df}}^{\mathrm{Ch}}$, $|M[A]|_{\mathrm{Df}}^{\mathrm{Ch}} = |M|_{\mathrm{Df}}^{\mathrm{Ch}}[A]$; and similarly defined for the rest cases.

2.3 Basic properties of the systems

▶ Proposition 1 (Uniqueness of types for Church-style).
If $\Gamma \vdash_{\mathrm{Ch}} M : A_1$ and $\Gamma \vdash_{\mathrm{Ch}} M : A_2$ then we have $A_1 \equiv A_2$.

▶ Proposition 2 (Erasure and lifting). Let $s, t \in \{\mathrm{Cu}, \mathrm{Tf}, \mathrm{Ha}, \mathrm{Df}, \mathrm{Ch}\}$ with $s < t$.

1. If $\Gamma \vdash_t M : A$ then $\Gamma \vdash_s |M|_s^t : A$.
2. If $\Gamma \vdash_s M : A$ then there exists a t-style $\lambda 2$-term N such that $|N|_s^t = M$ and $\Gamma \vdash_t N : A$.

▶ Proposition 3 (Generation lemma). Let $s \in \{\mathrm{Tf}, \mathrm{Df}, \mathrm{Ha}, \mathrm{Ch}\}$.

1. If $\Gamma \vdash_s x : A$ then $\Gamma(x) = A$.
2. If $\Gamma \vdash_s \lambda x : A_0.M : A_1$ then $\Gamma, x : A_0 \vdash_s M : A_2$ and $A_1 = (A_0 \to A_2)$ for some A_2, provided that $s \geq \mathrm{Ha}$.
3. If $\Gamma \vdash_s \lambda x.M : A_1$ then $\Gamma, x : A_0 \vdash_s M : A_2$ and $A_1 = (A_0 \to A_2)$ for some A_0, A_2, provided that $s \leq \mathrm{Df}$.
4. If $\Gamma \vdash_s M_1 M_2 : A_1$ then $\Gamma \vdash_s M_1 : A_0 \to A_1$ and $\Gamma \vdash_s M_2 : A_0$ for some A_0.
5. If $\Gamma \vdash_s \Lambda X.M : A_1$ then $\Gamma \vdash_s M : A_2$ and $A_1 = \forall X.A_2$ together with $X \notin \mathrm{FV}(\Gamma)$ for some A_2, provided that $s > \mathrm{Tf}$.
6. If $\Gamma \vdash_{\mathrm{Tf}} \Lambda.M : A_1$ then $\Gamma \vdash_{\mathrm{Tf}} M : A_2$ and $A_1 = \forall X.A_2$ together with $X \notin \mathrm{FV}(\Gamma)$ for some A_2.
7. If $\Gamma \vdash_s M[A] : A_1$ then $\Gamma \vdash_s M : \forall X.A_2$ and $A_1 = A_2[X := A]$ for some A_2, provided that $s \geq \mathrm{Df}$.
8. If $\Gamma \vdash_s M[] : A_1$ then $\Gamma \vdash_s M : \forall X.A_2$ and $A_1 = A_2[X := A]$ for some A, A_2, provided that $s \leq \mathrm{Ha}$.

Remarked that similar generation lemma holds for Curry-style $\lambda 2$, see [2, 32].

2.4 Type-related problems and relations between problems

▶ **Definition 4** (Type-related problems parameterized with styles)**.**

1. Type checking problem of s-style terms denoted by $\mathrm{TCP}(s)$:
   Given an s-style λ-term M, a type A, and a context Γ, determine whether $\Gamma \vdash_s M : A$.
2. Type inference problem of s-style λ-terms denoted by TIP(s):
   Given an s-style λ-term M and a context Γ, determine whether $\Gamma \vdash_s M : A$ for some type A.
3. Typability problem of s-style terms denoted by TPP(s):
   Given an s-style λ-term M, determine whether $\Gamma \vdash_s M : A$ for some context Γ and type A.

We show relations between type-related problems. For instance, if $\mathrm{TCP}(s)$ is reduced to $\mathrm{TIP}(s)$, then we write $\mathrm{TCP}(s) \hookrightarrow \mathrm{TIP}(s)$ for this. We write $\mathrm{TCP}(s) \longleftrightarrow \mathrm{TIP}(s)$ for both $\mathrm{TCP}(s) \hookrightarrow \mathrm{TIP}(s)$ and $\mathrm{TIP}(s) \hookrightarrow \mathrm{TCP}(s)$.

▶ Proposition 4 (Reductions between type-related problems).

1. $\mathrm{TCP}(s) \hookrightarrow \mathrm{TIP}(s)$ for $s \in \{\mathrm{Cu}, \mathrm{Tf}, \mathrm{Df}, \mathrm{Ha}, \mathrm{Ch}\}$.
2. $\mathrm{TIP}(s) \hookrightarrow \mathrm{TCP}(s)$ for $s \in \{\mathrm{Cu}, \mathrm{Tf}, \mathrm{Df}, \mathrm{Ha}\}$.
3. $\mathrm{TIP}(s) \hookrightarrow \mathrm{TPP}(s)$ for $s \in \{\mathrm{Df}, \mathrm{Ha}, \mathrm{Ch}\}$.
4. $\mathrm{TPP}(s) \hookrightarrow \mathrm{TIP}(s)$ for $s \in \{\mathrm{Cu}, \mathrm{Tf}, \mathrm{Df}\}$.

Proof. We show only the case of 3 where $s = \mathrm{Df}$ here.

3. Let $\Gamma = \{a_1 : A_1, \ldots, a_n : A_n\}$. $\Gamma \vdash_s M : B$ for some B if and only if $\Sigma \vdash_s M_0 : B$ for some B and some Σ, where z_0, z_1, z, y, Y are fresh variables, and
   $M_0 = z_0(z_1(z[\forall X.X]))(z_1 z)\ (z[(A_1 \to \cdots \to A_n \to Y) \to Y](\lambda a_1 \ldots \lambda a_n.yM))$.

Suppose that $\Gamma \vdash_s M : B$ for some B. Then M_0 is typable under some context Σ such as $\Sigma(z) = \forall X.X$.
In turn, if M_0 is typable then type of z should be a universal type, to say, $\forall X.\mathsf{F}(X)$, where F is a second-order variable with arity 1. From consistent typability of the two occurrences of z_1, we have the following unification equation: $\mathsf{F}(\forall X.X) \doteq \forall X.\mathsf{F}(X)$. Observe that the only solution to the unification equation is $[\mathsf{F} := (X \mapsto X)]$, i.e., the identity function, which implies that type of z is $\forall X.X$. Hence, we can recover the context Γ. ◀

3 Hole-application $\lambda 2$

We show that typability problems for hole-application $\lambda 2$ are undecidable. Next, in order to show decidability of type checking and type inference problems for hole-application $\lambda 2$, we provide a sound and complete algorithm for type inference. First, inference rules for hole-application $\lambda 2$ are listed in the following:

$$\frac{}{\Gamma \vdash_{\mathrm{hole}} x : \Gamma(x)}\ (\mathrm{var})$$

$$\frac{\Gamma, x{:}A_1 \vdash_{\mathrm{hole}} M : A_2}{\Gamma \vdash_{\mathrm{hole}} \lambda x{:}A_1.M : A_1 \to A_2}\ (\to I) \qquad \frac{\Gamma \vdash_{\mathrm{hole}} M_1 : A_1 \to A_2 \quad \Gamma \vdash_{\mathrm{hole}} M_2 : A_1}{\Gamma \vdash_{\mathrm{hole}} M_1 M_2 : A_2}\ (\to E)$$

$$\frac{\Gamma \vdash_{\mathrm{hole}} M : A}{\Gamma \vdash_{\mathrm{hole}} \Lambda X.M : \forall X.A}\ (\forall I)^{\star} \qquad \frac{\Gamma \vdash_{\mathrm{hole}} M : \forall X.A}{\Gamma \vdash_{\mathrm{hole}} M[] : A[X := B]}\ (\forall E)$$

3.1 TPP for hole-application $\lambda 2$

In order to show that TPP(Ha) is undecidable, we first introduce a restricted version of second-order unification, called a flat form [12], which can fit type constraints induced from hole-application terms. Then the undecidable unification problem is reduced to TPP(Ha) for hole-application $\lambda 2$. Although this reduction method is similar to that used in the previous work [12, 13, 14], we introduce the flat form and the encoding here to make the paper self-contained.

3.2 Second-order unification in flat form

We define expressions for unification problems. For this, the set of type variables is divided into three countable subsets: the set of first-order variables $\mathcal{V}_1$, the set of second-order functional variables $\mathcal{V}_2$, and the set of first-order constants $\mathcal{C}$. Then unification expressions are defined from first-order variables denoted by X and constants denoted by C, together with a binary constant $\to$ and second-order functional variables $\mathsf{F}^{(n)}A_1 \cdots A_n$ with arity n. The set of first-order expressions is denoted by $\mathcal{UE}_1$, and the expressions of first-order part are written by A, B as follows:

$A, B \in \mathcal{UE}_1 ::= X \mid C \mid (A \to B)$.

The sets of variables, constants, and sub-expressions in unification expressions are defined respectively as follows:

- $\mathrm{UVar}(X) = \{X\}$, $\mathrm{UVar}(C) = \emptyset$, $\mathrm{UVar}(A \to B) = \mathrm{UVar}(A) \cup \mathrm{UVar}(B)$, $\mathrm{UVar}(\mathsf{F}^{(n)}A_1 \cdots A_n) = \{\mathsf{F}^{(n)}\}$.
- $\mathrm{UCon}(X) = \emptyset$, $\mathrm{UCon}(C) = \{C\}$, $\mathrm{UCon}(A \to B) = \mathrm{UCon}(A) \cup \mathrm{UCon}(B)$, $\mathrm{UCon}(\mathsf{F}^{(n)}A_1 \cdots A_n) = \emptyset$.
- $\mathrm{UExp}(X) = \{X\}$, $\mathrm{UExp}(C) = \{C\}$, $\mathrm{UExp}(A \to B) = \{A \to B\} \cup \mathrm{UExp}(A) \cup \mathrm{UExp}(B)$, $\mathrm{UExp}(\mathsf{F}^{(n)}A_1 \cdots A_n) = \emptyset$.

The set E of unification equations in flat form is defined as follows:
$E ::= \emptyset \mid \{A \doteq B\} \cup E$
$\mid \{\mathsf{F}X_1 \dots X_n \doteq X \to A_1 \to \dots \to A_n \to o, \mathsf{F}C_1 \dots C_n \doteq X' \to C_1 \to \dots \to C_n \to o\} \cup E$,
provided that F is a functional variable with arity $n \geq 1$, X, X', X_i are fresh 1st-order variables, $C_1, \dots, C_n$ are new and pair wise distinct constants appeared nowhere else, o is a distinguished constant, and $\mathrm{UVar}(A_1, \dots, A_n) = \emptyset$.

We remark that each argument of a functional variable is restricted so that they are all 1st-order variables or pair wise distinct constants. Moreover, an equation with a functional variable always consists in such a pair of two equations, and functional variables in flat form appear only in this way.

Let E be a finite set of unification equations in flat form. Then $\mathrm{UVar}(E)$, $\mathrm{UCon}(E)$, and $\mathrm{UExp}(E)$ are naturally defined as well. A substitution is a partial function from the set of variables of unification expressions $\mathcal{V}_1 \cup \mathcal{V}_2$ to $\mathcal{UE}_1 \cup \{(X_1, \dots, X_n) \mapsto A \mid A \in \mathcal{UE}_1\}$. Let S, S_1, and S_2 range over the set of substitutions. A substitution S is naturally extended into a function S' from $\mathcal{UE}_1 \cup \mathcal{V}_2$ to $\mathcal{UE}_1 \cup \{(X_1, \dots, X_n) \mapsto A \mid A \in \mathcal{UE}_1\}$, such that
$S'(X) = S(X)$, $S'(C) = C$, $S'(A \to B) = S'(A) \to S'(B)$, and
$S'(\mathsf{F}A_1 \dots A_n) = B[X_1 := S'(A_1), \dots, X_n := S'(A_n)]$,
where F is a second-order variable with arity n and $S(\mathsf{F}) = (X_1, \dots, X_n) \mapsto B$ for $B \in \mathcal{UE}_1$. We may write simply S for S'. An instance E is solvable if there exists a substitution S such that $S(A) = S(B)$ and $S(\mathsf{F}A_1 \dots A_n) = S(B')$ for all unification equations in E in the form of either $A \doteq B$ or $\mathsf{F}A_1 \dots A_n \doteq B'$.

▶ Proposition 5 ([12]). The second-order unification problem in flat form is undecidable.

3.3 Reduction from flat form to TPP(Ha)

For encoding an instance of second-order unification E in flat form, we assume one-to-one mappings between unification expressions and term variables of $\lambda 2$. Based on this, we write x_A, y_A for $A \in \mathrm{UExp}(B)$ where $B \in \mathcal{UE}_1$, and $x_\mathsf{F}, y_\mathsf{F}$ for $\mathsf{F} \in \mathcal{V}_2$. In particular, the distinguished constant $o \in \mathcal{C}$ provides x_o, y_o, y_{o_2}, and so on. We write $o^k \to o$ for type $(o \to (\dots \to (o \to o)))$ with $(k+1)$-times o. As a shorthand, we define $\lambda 2$-terms such that $M[]^{n+1} = (M[])[]^n$, $M[]^0 = M$.

▶ **Definition 5** (Encoding of unification expressions). **1.** Case E of $\emptyset$: $[\![E]\!] = x_o$
2. Case E of $\{A \doteq B\} \cup E_0$: $[\![E]\!] = y_{o_4}\ (y_A x_B)\ [\![A]\!]\ [\![B]\!]\ [\![E_0]\!]$
3. Case E of $E_f \cup E_0$, where $E_f = \{\mathsf{F}X_1 \dots X_n \doteq B_1, \mathsf{F}C_1 \dots C_n \doteq B_2\}$ together with $B_1 = (X \to A_1 \to \dots \to A_n \to o)$ and $B_2 = (X' \to C_1 \to \dots \to C_n \to o)$:

$$\begin{aligned}[\![E]\!] = {} & y_{o_9}\ (y_\mathsf{F} x_\mathsf{F})\ (y_\mathsf{F}(\Lambda Z_1 \dots \Lambda Z_n.(\Lambda Z.\lambda z{:}Z.\lambda z_1{:}Z_1 \dots \lambda z_n{:}Z_n.x_o)[])) \\ & (y_{B_1}(x_\mathsf{F}[]^n))\ (y_{B_1}((\Lambda Z.\lambda z{:}Z.\lambda z_1{:}A_1 \dots \lambda z_n{:}A_n.x_o)[])) \\ & (y_{B_2}(x_\mathsf{F}[]^n))\ (y_{B_2}((\Lambda Z'.\lambda z'{:}Z'.\lambda z_1{:}C_1 \dots \lambda z_n{:}C_n.x_o)[])) \\ & [\![B_1]\!]\ [\![B_2]\!]\ [\![E_0]\!]\end{aligned}$$

4. For $A \in \mathcal{UE}_1$, $[\![A]\!]$ is defined as follows:
a. $[\![X]\!] = y_{o_1}(y_X x_X)$
b. $[\![C]\!] = y_{o_1}(y_C x_C)$
c. $[\![A \to B]\!] = y_{o_4}\ (y_B(x_{A\to B} x_A))\ (y_{A\to B} x_{A\to B})\ [\![A]\!]\ [\![B]\!]$
5. $\Sigma_\Delta = \Sigma_o \cup \Sigma(\Delta)$, where Σ_o and $\Sigma(\Delta)$ are defined as follows for $\Delta = A$ or E:
a. $\Sigma_o = \{x_o : o, y_{o_1} : o \to o, y_{o_2} : o \to o \to o, \dots, y_{o_k} : o^k \to o\}$ for $k = 9$
b. $\Sigma(\Delta) = \{x_C : C, y_C : C \to o \mid C \in \mathrm{UCon}(\Delta)\}$

An idea on encoding of first-order follows the structure of expressions, such that a unification expression provides an λ-term consisting of consecutive application of variables associated to each sub-expression, which induces type constraints leading to substitutions for the unification expression. An idea on encoding of second-order is such that a term variable x_{F} associated to a functional variable F of unification should have a universal type, whose instance by application of $(\forall E)$ must be equivalent to a substitution instance of the right-hand side B of the corresponding unification equation $\mathsf{F}(\cdots) \doteq B$.

▶ **Lemma 6.** *Let $A \in \mathcal{UE}_1$, $\Sigma_A = \Sigma_o \cup \Sigma(A)$, and S be a substitution from $\mathcal{UE}_1$ to $\mathcal{UE}_1$. Then we have $\Sigma_A, \Gamma \vdash_{\mathrm{hole}} [\![A]\!] : o$ for some context Γ such that $\Gamma(x_B) = S(B)$ and $\Gamma(y_B) = (S(B) \to o)$ for each $B \in \mathrm{UExp}(A)$.*

Proof. We remark that Γ should declare statements for all free variables x_B, y_B in $[\![A]\!]$ where $B \in \mathrm{UExp}(A)$. By induction on the structure of A. We show one case here.

1. Case of $A = (A_1 \to A_2)$:
   From the induction hypotheses, we have $\Sigma_{A_1}, \Gamma_1 \vdash_{\mathrm{hole}} [\![A_1]\!] : o$ and $\Sigma_{A_2}, \Gamma_2 \vdash_{\mathrm{hole}} [\![A_2]\!] : o$, such that $\Gamma_1(x_{B_1}) = S(B_1)$, $\Gamma_1(y_{B_1}) = S(B_1) \to o$ for each $B_1 \in \mathrm{UExp}(A_1)$ and $\Gamma_1(x_{B_2}) = S(B_2)$, $\Gamma_1(y_{B_2}) = S(B_2) \to o$ for each $B_2 \in \mathrm{UExp}(A_2)$. Then we can merge Γ_1 and Γ_2 into Γ so that $\Gamma(x_{A_1 \to A_2}) = S(A_1) \to S(A_2)$ and $\Gamma(y_{A_1 \to A_2}) = \Gamma(x_{A_1 \to A_2}) \to o$, since $\Gamma_1(x_B) = S(B) = \Gamma_2(x_B)$ for $B \in \mathrm{UExp}(A_1) \cap \mathrm{UExp}(A_2)$. Hence, We have $\Sigma_A, \Gamma \vdash_{\mathrm{hole}} [\![A_1 \to A_2]\!] : o$ for some Γ with the desired property. ◀

▶ Proposition 6. A flat form E is solvable if and only if $\Sigma_E, \Gamma \vdash_{\mathrm{hole}} [\![E]\!] : o$ for some Γ.

Proof. The only-if part can be verified so that Γ is given by a unifier of E. We show here the if-part. Suppose that the encoding $[\![E]\!]$ has type o under Σ_E and some context Γ. From consistent type of the encoding of first-order equations $A' \doteq B' \in E$, we have $\Gamma(x_{A'}) = \Gamma(x_{B'})$. From this and Lemma 6, we can define a substitution S for first-order variables in $\mathrm{UVar}(E)$ such that $S(A') = \Gamma(x_{A'}) = \Gamma(x_{B'}) = S(B')$. Next, we verify a consistent type of the encoding of second-order equations. Considering the first and second arguments of y_{o_9}, the term x_{F} has type $\forall Z_1 \ldots \forall Z_n.(A \to Z_1 \to \cdots \to Z_n \to o)$ for some A. Here, we can assume that A should contain no quantifiers $\forall$, since the type A is simply related to the first argument type of x_{B_1} and x_{B_2}. Even if A contained for instance $\forall Y.B$, then one could replace this with $B[Y := Y']$ using a fixed type variable Y'. Then, from consistent type of the three occurrences of each argument of y_{B_1}, the three terms, x_{B_1}, $x_{\mathsf{F}}[]^n$, and $(\Lambda Z.\lambda z{:}Z.\lambda z_1{:}A_1 \ldots \lambda z_n{:}A_n.x_o)[]$, all have the same type $(A[Z_1 := A_1, \ldots, Z_n := A_n] \to A_1 \to \cdots \to A_n \to o)$. Following a similar pattern, the three arguments of y_{B_2} are also well-typed. Therefore, a flat form E becomes solvable under a substitution S such that

$S(\mathsf{F}) = (Z_1, \ldots, Z_n) \mapsto (A \to Z_1 \to \cdots \to Z_n \to o)$, $S(X_i) = A_i$ for $1 \le i \le n$,
$S(X) = A[Z_1 := A_1, \ldots, Z_n := A_n]$, and $S(X') = A[Z_1 := C_1, \ldots, Z_n := C_n]$. ◀

▶ **Theorem 7** (TPP for hole-application $\lambda 2$)**.** TPP *is undecidable for hole-application $\lambda 2$.*

Proof. A flat form E is solvable

iff $\Gamma, \Sigma_E \vdash_{\mathrm{hole}} [\![E]\!] : o$ for some Γ by Proposition 6
iff $\Gamma \vdash_{\mathrm{hole}} \lambda v{:}\forall X.(X \to o).v[](\lambda \vec{z}{:}\Sigma_E(\vec{z}).[\![E]\!]) : A$ for some A and some Γ.

Here, we write $\lambda \vec{z} : \Sigma_E(\vec{z}).[\![E]\!]$ for $\lambda z_1 : \Sigma_E(z_1) \ldots \lambda z_n : \Sigma_E(z_n).[\![E]\!]$ with $\{z_1, \ldots, z_n\} = \mathrm{dom}(\Sigma_E)$. ◀

3.4 TCP and TIP for hole-application $\lambda 2$

From Proposition 4, the problems TCP(Ha) and TIP(Ha) are equivalent, and we provide a type inference algorithm $\mathsf{type}(\Gamma; M)$, which computes a type of a hole-application term M under a context Γ. The algorithm involves a unification procedure, for which we introduce

new type variables called unification variables consisting of first-order variables denoted by α, β and functional variables denoted by F. For the technical reason, the following syntax $\hat{A} \in \mathsf{Uexp}$ is defined from types and first-order unification variables:

$\hat{A} \in \mathsf{Uexp} ::= X \mid \alpha \mid (\hat{A} \to \hat{A}) \mid \forall X.\hat{A}$

A substitution denoted by S used in unification should operate only on unification variables, such that $S(\alpha), S(\mathsf{F})(\beta) \in \mathsf{Uexp}$ and $S(X) = X$, i.e., the domain of substitutions is the set of unification variables, and the range is Uexp.

▶ **Definition 8** (Type inference algorithm type)**.**

1. $\mathsf{type}(\Gamma; x) = \Gamma(x)$
2. $\mathsf{type}(\Gamma; \lambda x\!:\!A.M) = (A \to \mathsf{type}(\Gamma, x\!:\!A; M))$
3. $\mathsf{type}(\Gamma; MN) =$
   let $\hat{B}_1 = \mathsf{type}(\Gamma; M)$ and (* α is a fresh unification variable *)
   let $\hat{B}_2 = \mathsf{type}(\Gamma; N)$ and $S = \mathsf{unify}(\hat{B}_1, \hat{B}_2 \to \alpha)$ in $S(\alpha)$
4. $\mathsf{type}(\Gamma; \Lambda X.M) =$ (* S is an arbitrary substitution for unification variables *)
   let $\hat{B} = \mathsf{type}(\Gamma; M)$ and $X \notin \mathrm{FV}(\Gamma)$ in $\forall X.S(\hat{B})$
5. $\mathsf{type}(\Gamma; M[]) =$
   let $\hat{B} = \mathsf{type}(\Gamma; M)$ and $S = \mathsf{unify}(\forall X.\mathsf{F}(X), \hat{B})$ (* β is a fresh unification variable, in $S(\mathsf{F})(\beta)$ and F is a fresh functional unification variable with arity 1 *)

We remark that second-order unification used in the algorithm is a special case of patterns unification of Miller [23], such that arguments of a functional variable are distinct bound variables in expressions. Since unification of patterns is decidable and gives a most general unifier if unifiable [23], the unification problem such as $\mathsf{unify}(\forall X.\mathsf{F}(X), \hat{B})$ is decidable. Hence, the unification procedure returns the most general solution to the type inference problem. In this sense, `type` gives rise to a decidable sub-language of which is derived by the general translation V of Pfenning [26] in the case of the omega-order calculus F_ω.

Let $\bot \equiv \forall X.X$. We show an example of $\mathsf{type}(\langle\rangle; \Lambda Z.\lambda x\!:\!\bot.x[]x)$ in the following:

1. $\mathsf{type}(x\!:\!\bot; x[]) = (X \mapsto X)\beta = \beta$ for a fresh unification variable β,
   where $\mathsf{unify}(\forall X.\mathsf{F}(X), \bot) = [\mathsf{F} := (X \mapsto X)]$.
2. $\mathsf{type}(x\!:\!\bot; x[]x) = \alpha$ for a fresh unification variable α,
   where $\mathsf{unify}(\beta, \bot \to \alpha) = [\beta := (\bot \to \alpha)]$.
3. $\mathsf{type}(\langle\rangle; \lambda x\!:\!\bot.x[]x) = (\bot \to \mathsf{type}(x\!:\!\bot; x[]x)) = (\bot \to \alpha)$
4. $\mathsf{type}(\langle\rangle; \Lambda Z.\lambda x\!:\!\bot.x[]x) = \forall Z.\mathsf{s}(\mathsf{type}(\langle\rangle; \lambda x\!:\!\bot.x[]x)))$
   $= \forall Z.(\bot \to \mathsf{s}(\alpha))$ for an arbitrary substitution s.

In addition, we show a proof figure below, which provides a type for the term. Although the term may have yet another type, all possible types for $\Lambda Z.\lambda x\!:\!\bot.x[]x$ can be expressed by the inferred type $\mathsf{type}(\langle\rangle; \Lambda Z.\lambda x\!:\!\bot.x[]x)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{x\!:\!\bot \vdash_{\mathrm{hole}} x : \bot}{x\!:\!\bot \vdash_{\mathrm{hole}} x[] : \bot \to Z}\,(\forall E) \qquad x\!:\!\bot \vdash_{\mathrm{hole}} x : \bot}{x\!:\!\bot \vdash_{\mathrm{hole}} x[]x : Z}\,(\to E)}{\vdash_{\mathrm{hole}} \lambda x\!:\!\bot.x[]x : \bot \to Z}\,(\to I)}{\vdash_{\mathrm{hole}} \Lambda Z.\lambda x\!:\!\bot.x[]x : \forall Z.(\bot \to Z)}\,(\forall I)^{\star}$$

One of the points of the algorithm is that any type to be filled into a hole [] can be represented by a unification variable, which is handled by a decidable fragment of second-order unification. Another point is that a universal type of a term $\Lambda X.M$ should be in the form of $\forall X.S(\hat{A})$, where $\hat{A}$ is a type of M, and S is an arbitrary substitution for unification variables in $\hat{A}$. In the process of unification, such an arbitrary substitution is handled as delayed substitutions at an object level.

▶ Proposition 7 (Soundness and completeness of type).

1. If $\mathsf{type}(\Gamma; M) = \hat{A}$ then $\Gamma \vdash_{\mathrm{hole}} M : \hat{A}$.
2. Given a context Γ and a term M, let A be a type such that $\Gamma \vdash_{\mathrm{hole}} M : A$. Then we have $\mathsf{type}(\Gamma; M) = \hat{B}$ such that $A = S(\hat{B})$ under some substitution S for unification variables.

Proof. A type system for hole-application $\lambda 2$ to handle $\hat{A}$ can be naturally introduced, such that infer $\Gamma \vdash_{\mathrm{hole}} M[] : A[X := \hat{B}]$ from $\Gamma \vdash_{\mathrm{hole}} M : \forall X.A$. We claim that if $\Gamma \vdash_{\mathrm{hole}} M : \hat{A}$ then $\Gamma \vdash_{\mathrm{hole}} M : S(\hat{A})$ for any substitution S for unification variables.

In the following, we show some the cases here. The algorithm is proved to be sound by induction on the structure of M.

1-1. Case of $\mathsf{type}(\Gamma; M[]) = S(\mathsf{F})(\beta)$, where $S = \mathsf{unify}(\forall X.\mathsf{F}(X), \mathsf{type}(\Gamma; M))$:
From the induction hypothesis, we have $\Gamma \vdash M : \mathsf{type}(\Gamma; M)$, and then $\Gamma \vdash M : S(\mathsf{type}(\Gamma; M))$ where $S(\mathsf{type}(\Gamma; M)) = \forall X.S(\mathsf{F})(X)$. Hence, we establish that $\Gamma \vdash M[] : S(\mathsf{F})(\beta)$ where β is a fresh unification variable.

1-2. Case of $\mathsf{type}(\Gamma; \Lambda X.M) = \forall X.S(\mathsf{type}(\Gamma; M))$ for any S, where $X \notin \mathrm{FV}(\Gamma)$:
From the induction hypothesis, we have $\Gamma \vdash M : \mathsf{type}(\Gamma; M)$, and then $\Gamma \vdash M : S(\mathsf{type}(\Gamma; M))$ for any substitution S for unification variables. Hence, $\Gamma \vdash \Lambda X.M : \forall X.S(\mathsf{type}(\Gamma; M))$.

The completeness property is proved by induction on the derivation of $\Gamma \vdash_{\mathrm{hole}} M : A$.

2-1. $\Gamma \vdash \Lambda X.M : \forall X.A$ from $\Gamma \vdash M : A$, where $X \notin \mathrm{FV}(\Gamma)$:
From the induction hypothesis, we have $\mathsf{type}(\Gamma; M) = \hat{A}_1$ where $A = S(\hat{A}_1)$ for some S. Then we confirm that $\forall X.A = \forall X.S(\hat{A}_1) = \mathsf{type}(\Gamma; \Lambda X.M)$.

2-2. $\Gamma \vdash M[] : A[X := B]$ from $\Gamma \vdash M : \forall X.A$:
From the induction hypothesis, we have $\mathsf{type}(\Gamma; M) = \hat{A}_1$ and $\forall X.A = S(\hat{A}_1)$ for some S. Then we have a unifier $S = \mathsf{unify}(\forall X.\mathsf{F}(X), \hat{A}_1)$, since $\forall X.S(\mathsf{F})(X) = S(\hat{A}_1) = \forall X.A$ where $S(\mathsf{F})(X) = A$. Hence, $A[X := B] = S(\mathsf{F})(B) = S(\mathsf{F})(\beta)[\beta := B] = S_1(\mathsf{type}(\Gamma; M[]))$ for some S_1, such that $S_1 = S \cup \{[\beta := B]\}$ where β is a fresh unification variable. ◀

3.5 Subject reduction of hole-application $\lambda 2$

We define reduction rules for hole-application terms. The idea is to introduce a fresh and distinguished type variable at each reduction of type variable abstraction. Then, from a typing derivation, we can extract a concrete type, by which the fresh type variable should be replaced.

▶ **Definition 9** (Reduction rules for hole-application $\lambda 2$)**.**

(β) $(\lambda x\!:\!A.M)N \to M[x := N]$
(β_t) $(\Lambda X.M)[] \to M[X := \alpha]$ where α is a fresh type variable.

For instance, we have the following judgement: $\vdash_{\mathrm{hole}} \Lambda Y.(\Lambda X.\lambda x\!:\!X.x)[] : \forall Y.((Y \to Y) \to Y \to Y)$. Then $\Lambda Y.(\Lambda X.\lambda x : X.x)[] \to \Lambda Y.\lambda x : \alpha.x$ where α is a fresh type variable. Now, from a derivation of the judgement:

$$\dfrac{\dfrac{\dfrac{\dfrac{x\!:\!X \vdash x : X}{\vdash \lambda x\!:\!X.x : X \to X}\,(\to I)}{\vdash \Lambda X.\lambda x\!:\!X.x : \forall X.(X \to X)}\,(\forall I)^{\star}}{\vdash (\Lambda X.\lambda x\!:\!X.x)[] : (Y \to Y) \to Y \to Y}\,(\forall E)}{\vdash \Lambda Y.(\Lambda X.\lambda x\!:\!X.x)[] : \forall Y.((Y \to Y) \to Y \to Y)}\,(\forall I)^{\star}$$

a replacement for α can be extracted such that $R(\alpha) = (Y \to Y)$. Note that this replacement should not be called a substitution, since free Y in $R(\alpha)$ is to be in the scope of ΛY of the example $\Lambda Y.\lambda x\!:\!R(\alpha).x$.

▶ Proposition 8 (Subject reduction). If $\Gamma \vdash_{\mathrm{hole}} M : A$ and $M \to N$, then $\Gamma \vdash_{\mathrm{hole}} R(N) : A$ for some replacement R for fresh variables.

Proof. By induction on the derivation $M \to N$. We show some of the interesting cases.

1. $\Gamma \vdash (\Lambda X.M)[] : A$ and $(\Lambda X.M)[] \to M[X := \alpha]$:
   From the generation lemma, we have $\Gamma \vdash \Lambda X.M : \forall X.A'$ where $A = A'[X := B]$ and $\Gamma \vdash M : A'$ where $X \notin \mathrm{FV}(\Gamma)$ for some A', B. Then we have $\Gamma \vdash M[X := B] : A'[X := B]$. Hence, $\Gamma \vdash R(M[X := \alpha]) : A'[X := B]$ for some replacement R such that $R(\alpha) = B$.
2. $\Gamma \vdash M[] : A$ and $M[] \to M_1[]$:
   From the generation lemma, we have $\Gamma \vdash M : \forall X.A'$ with $A = A'[X := B]$ for some A', B. From the induction hypothesis w.r.t. $\Gamma \vdash M : \forall X.A'$ and $M \to M_1$, we have $\Gamma \vdash R(M_1) : \forall X.A'$ for some R, and hence $\Gamma \vdash R(M_1)[] : A'[X := B]$. ◀

Finally, we extend the idea of hole-application to an omega-order system F_ω.

4 Hole-application F_ω

We introduce a formal system of hole-application F_ω. The system consists of kinds K, type constructors A, hole-application terms M, and contexts Γ. For a kind K, an order $\mathsf{ord}(K)$ is defined, such that $\mathsf{ord}(\star) = 2$ and $\mathsf{ord}(K_1 \to \cdots \to K_n \to \star) = \max\{\mathsf{ord}(K_i) \mid 1 \le i \le n\} + 1$. A fragment of F_ω restricted to $K = \star$, i.e., $\mathsf{ord}(K) = 2$, coincides with $\lambda 2$.

Compared with hole-application $\lambda 2$, hole-application F_ω has a hole $[]_K$ with a kind K, which is to be filled with a type constructor of kind K, see the inference rule (ΠE) in the following. Such a hole has already been introduced in Pfenning [26].

▶ **Definition 10** (Hole-application F_ω).

1. Kinds
$$K ::= \star \mid (K \to K)$$
2. Type constructors
$$A ::= X \mid (A \to A) \mid \Pi X\!:\!K.A \mid \Lambda X\!:\!K.A \mid AA$$
3. Hole-application terms
$$M ::= x \mid \lambda x\!:\!A.M \mid MM \mid \Lambda X\!:\!K.M \mid M[]_K$$
4. Contexts
$$\Gamma ::= \langle\rangle \mid X\!:\!K, \Gamma \mid x\!:\!A, \Gamma$$

Next we define inference rules for well-formed contexts, well-formed kinds, well-formed elements of a kind, and well-formed elements of a type, respectively. Here, we show rules only for well-formed elements of a type.

1. Well-formed elements of a type:
$$\frac{\vdash \Gamma \quad x\!:\!A \in \Gamma}{\Gamma \vdash x : A}\ (\mathrm{var})$$
$$\frac{\Gamma \vdash A_1 : \star \quad \Gamma, x\!:\!A \vdash M : A_2}{\Gamma \vdash (\lambda x\!:\!A_1.M) : (A_1 \to A_2)}\ (\to I) \qquad \frac{\Gamma \vdash M_1 : (A_1 \to A_2) \quad \Gamma \vdash M_2 : A_1}{\Gamma \vdash M_1 M_2 : A_2}\ (\to E)$$

$$\frac{\Gamma \vdash K \quad \Gamma, X{:}K \vdash M : A}{\Gamma \vdash (\Lambda X{:}K.M) : (\Pi X{:}K.A)}\ (\Pi I) \qquad \frac{\Gamma \vdash M : (\Pi X{:}K.A_1) \quad \Gamma \vdash A_2 : K}{\Gamma \vdash M[]_K : A_1[X := A_2]}\ (\Pi E)$$

$$\frac{\Gamma \vdash M : A_1 \quad \Gamma \vdash A_2 : \star \quad A_1 =_{\beta\eta} A_2}{\Gamma \vdash M : A_2}\ (\mathrm{conv})$$

Even in the case of omega-order, the two problems TCP(Ha) and TIP(Ha) are equivalent each other as proved by Proposition 4. Toward type inference for hole-application F_ω, type constructors are extended with fresh type variables called unification variables denoted by $\alpha, \beta, \mathsf{F}, \mathsf{G}$, as follows:

$\hat{A} ::= X \mid \alpha \mid (\hat{A} \to \hat{A}) \mid \Pi X{:}K.\hat{A} \mid \Lambda X{:}K.\hat{A} \mid \hat{A}\hat{A}$.

Here, we show that TIP and TCP for hole-application F_ω are undecidable. For this, we give a reduction from higher-order unification [16, 15] to TCP for hole-application F_ω.

The theory of simply type λ-calculus is defined as usual, but in terms of type constructors of F_ω. Here, we assume the following variable conventions: F, G for free variables, and X, Y, Z for constants or bound variables. In addition, $\star$ stands for an atomic type.

- Terms

$$t, s ::= X \mid \mathsf{F} \mid \Lambda X.t \mid (t\ s)$$

- Types

$$K ::= \star \mid (K \to K)$$

Given a well-typed term t of simply typed λ-calculus, then define a type constructor $t^\sharp$ of hole-application F_ω as follows, where a free variable F will be interpreted as a bound variable such as $\Pi\mathsf{F}{:}K.(\cdots)$ in F_ω.

1. $X^\sharp = X$
2. $\mathsf{F}^\sharp = \mathsf{F}$
3. $(\Lambda X.t)^\sharp = \Lambda X^\sharp{:}K.t^\sharp$, where X has type K
4. $(t\ s)^\sharp = t^\sharp\ s^\sharp$

Given an instance $s \doteq t$ of higher-order unification, where $\{\mathsf{F}_1 : L_1, \ldots, \mathsf{F}_n : L_n\} = \mathrm{FV}(s)$ together with type L_i for each free variable F_i in s, $\{\mathsf{G}_1 : L'_1, \ldots, \mathsf{G}_m : L'_m\} = \mathrm{FV}(t)$ with type L'_i for each free variable G_i in t, and the terms s and t both have type $(K_1 \to \cdots \to K_p \to \star)$. Then define a context $\Gamma_{s=t}$ of hole-application F_ω as follows:

$\{X_1 : K_1, \ldots, X_p : K_p, Z : \star,$
$\quad x_s : (\Pi\mathsf{F}_1{:}L_1 \ldots \Pi\mathsf{F}_n{:}L_n.(s^\sharp X_1 \ldots X_p \to Z)),\ x_t : (\Pi\mathsf{G}_1{:}L'_1 \ldots \Pi\mathsf{G}_m{:}L'_m.t^\sharp X_1 \ldots X_p)\}$

▶ Proposition 9 (TCP(hole-F_ω)). An instance of higher-order unification $s \doteq t$ is solvable if and only if $\Gamma_{s=t} \vdash x_s[]^n_{\vec{L}}(x_t[]^m_{\vec{L'}}) : Z$ in hole-application F_ω.

Proof. We show the if-part here. Suppose that $\Gamma_{s=t} \vdash x_s[]^n_{\vec{L}}(x_t[]^m_{\vec{L'}}) : Z$ in hole-application F_ω. Then we have the following judgements by a chain of applications of (ΠE):

$\Gamma_{s=t} \vdash x_s[]^n_{\vec{L}} : (s^\sharp X_1 \ldots X_p \to Z)[\mathsf{F}_1 := \alpha_1, \ldots, \mathsf{F}_n := \alpha_n]$, and
$\Gamma_{s=t} \vdash x_t[]^m_{\vec{L'}} : t^\sharp X_1 \ldots X_p[\mathsf{G}_1 := \beta_1, \ldots, \mathsf{G}_m := \beta_m]$,

where α_i, β_j are fresh type variables called unification variables with appropriate kinds. From consistent type of $x_s[]^n_{\vec{L}}(x_t[]^m_{\vec{L'}})$ under $\Gamma_{s=t}$, there exists a unifier for the unification equation:

$(s^\sharp X_1 \ldots X_p \to Z)[\mathsf{F}_1 := \alpha_1, \ldots, \mathsf{F}_n := \alpha_n] \doteq t^\sharp X_1 \ldots X_p[\mathsf{G}_1 := \beta_1, \ldots, \mathsf{G}_m := \beta_m] \to \alpha$,

where α is a unification variable with kind $\star$. That is, the following equation is solvable:

$s^\sharp X_1 \ldots X_p[\mathsf{F}_1 := \alpha_1, \ldots, \mathsf{F}_n := \alpha_n] \doteq t^\sharp X_1 \ldots X_p[\mathsf{G}_1 := \beta_1, \ldots, \mathsf{G}_m := \beta_m]$.

Hence, $s^\sharp \doteq t^\sharp$ is unifiable, and then exactly so is $s \doteq t$. ◀

▶ **Theorem 11** (TCP(hole-F_ω),TIP(hole-F_ω))**.** TCP(hole-F_ω) *and* TIP (hole-F_ω) *are equivalent and undecidable.*

Proof. From Propositions 4 and 9. ◀

Remarked that the use of kind-labels annotated to holes is not essential in the proof for undecidability. Since the context has $x_s : (\Pi \mathsf{F}_1 : L_1 \ldots \Pi \mathsf{F}_n : L_n.(s^\sharp X_1 \ldots X_n \to Z))$, we can apply (ΠE) to A only with kind L_1. In the next section, we will observe that another kind of labels is essential for undecidability of TCP(hole-$\lambda 2$) and TIP(hole-$\lambda 2$), contrary to this case of omega-order.

5 Concluding remarks and summary of results

The type-related problems (TCP, TIP, TPP) have been studied extensively from various viewpoints, e.g., type ranks [19, 20, 11], type levels [21, 13, 14], partially typed terms [5, 26, 27, 11]. Here, we discussed mainly from a perspective of type erasure mapping. We have examined three intermediate $\lambda 2$-terms between Church-style and Curry-style. In particular, TCP and TIP for hole-application $\lambda 2$-terms turn out to be decidable, providing a type inference algorithm that is sound and complete. The algorithm involves two important features: one is decidable second-order unification, which is a special case of patterns unification [23, 8], and another is delayed substitutions, which are employed to denote arbitrary substitutions at an object level. On the other hand, TPP(Ha) is undecidable for hole-application $\lambda 2$.

Next, we extended the idea of hole-application to F_ω, and proved that TCP and TIP then become undecidable for the system. Strictly speaking, the problems are undecidable for F_3 from undecidability of second-order unification [15].

We summarize the results on the type-related problems for $\lambda 2$. Table 1 shows the decidability results for $\lambda 2$ and relations on the type-related problems. Reduction relations $(\hookrightarrow, \longleftrightarrow, \hookleftarrow)$ between problems follow Proposition 4. To our knowledge, it is a new result that TCP, TIP, and TPP are all equivalent in the case of domain-free, which implies a corollary such that typability of domain-free $\lambda 2$ is undecidable [24]. While the table shows that TPP is undecidable for any style, TCP and TIP have the boundaries between hole-application and domain-free. Compared with Church-style, TIP remains decidable even after deleting polymorphic instance information on application of $(\forall E)$. However, on application of $(\to I)$, deleting polymorphic domains makes TIP undecidable. Following [11], finding out deleted polymorphic domains is to find a polymorphic context, which leads to undecidable unification (simple instances). Therefore, the introduction of hole-application reveals that polymorphic domains are considered as the most essential information for (un)decidable TIP.

We make some observations on our results from the viewpoint of partially typed terms.

Partial type reconstruction

Partially typed terms (preterms) are defined as follows:

$P ::= x \mid \lambda x{:}A.P \mid PP \mid \Lambda X.P \mid P[A] \mid \lambda x.P \mid P[]$

The problem of partial type reconstruction is a problem: given a context Γ and a preterm P, determine whether there exists a term M in Church-style such that $\Gamma \vdash_{\mathrm{Ch}} M : A$ and $|M| = P$ for some A. The problem has been studied extensively and proved to be, in general, undecidable by Boehm [5] and Pfenning [27]. Moreover, Pfenning [26] shows the precise correspondence such that the problem in the n-th order λ-calculus is equivalent to n-th order unification that is undecidable in general for $n \geq 2$. Along this line, partial type reconstruction problems for s-style terms can be defined naturally. Then TIP(s) (type inference for s-style terms) is equivalent to partial type reconstruction for s-style terms.

Some of intermediate structures, e.g., domain-free and type-free, are already known and investigated.

Domain-free style

Pure Type Systems [2] in domain-free style were studied in detail in Barthe and Sørensen [4]. Domain-free systems serve as a good source language for CPS-translation. For instance, domain-free $\lambda 2$ and $\lambda^{\exists}$ are demonstrated in [10]. Parigot's $\lambda\mu$-calculus [25] in domain-free style is investigated in [9] for call-by-value second-order language with control operators.

Type-free style

The type reconstruction problem for type-free style $\lambda 2$ was described in Pfenning [27] as an instance for terms completely devoid of types except for $[]$ and Λ, and this restricted problem had been open. Recently, a negative answer to the problem is proved in [13]. The type-free style gives a compact proof description, such that this style contains the complete information on which and where inference rules are applied *á la* Church-style, but no information on what types are involved in the rules *á la* Curry-style.

Partially typed terms with labels

Another interesting variant of $\lambda 2$-terms is partially typed terms together with labels [11]. Labels denoted by a are introduced into preterms as follows:

$P ::= x \mid \lambda x{:}[A]^a.P \mid PP \mid \Lambda X.P \mid P[A]^a \mid \lambda x{:}[]^a.P \mid P[]^a$

Here, the placeholder $[]^a$ indicates that a type has been erased, and moreover, the label a in $[]^a$ will be used to identify the occurrences of $[]$, i.e., the holes $[]$ with the same label should be obtained by erasing the same type.

Preterms with no labels can be translated to preterms with labels using fresh ones, such that $\lfloor \lambda x.M \rfloor = \lambda x{:}[]^a.\lfloor M \rfloor$ for a fresh label a and $\lfloor M[] \rfloor = \lfloor M \rfloor []^a$ for a fresh label a. Hence, $\Gamma \vdash P : A$ without labels iff $\Gamma \vdash \lfloor P \rfloor : A$ with labels, which implies that the type-related problems of preterms can be embedded into those with labels. For instance, the type-related problems of domain-free style with labels:

$M ::= x \mid \lambda x^a.M \mid MM \mid \Lambda X.M \mid M[A]$,

and the problems of type-free style with labels:

$M ::= x \mid \lambda x^a.M \mid MM \mid \Lambda.M \mid M[]^a$

are also undecidable. In addition, TPP of hole-application style with labels:

$M ::= x \mid \lambda x{:}A.M \mid MM \mid \Lambda X.M \mid M[]^a$

is undecidable by a reduction from TPP for Church-style [30] without labels, as follows:

$\Gamma \vdash_{\mathrm{Ch}} M : A$ for some Γ and some A if and only if $\Gamma \vdash_{\mathrm{hole}^a} \lceil M \rceil : A$ for some Γ and some A, where $\lceil M[A] \rceil = (\lambda v{:}(A \to A).\lceil M \rceil []^a)(\lambda x{:}A.(\Lambda X.\lambda y{:}X.y)[]^a x)$.

Although TCP and TIP for hole-application without labels are to be decidable in this paper, the two problems with labels become undecidable by a reduction from TPP of hole-application with labels, as follows: Let $\{x_1, \ldots, x_n\} = \mathrm{FV}(M)$.

$\Gamma \vdash_{\mathrm{hole}^a} M : A$ for some Γ, A iff $z_1 : \forall X.X, \ldots, z_n : \forall X.X \vdash_{\mathrm{hole}^a} \langle M \rangle : A$ for some A, where $\langle x_i \rangle = z_i []^{a_i}$ for a fresh variable z_i and a fresh label a_i.

Hence, the type checking problem for hole-application with labels becomes undecidable by Proposition 4. These observations mean the use of labels for hole-application $\lambda 2$ is essential for undecidability of TCP(hole-$\lambda 2$) and TIP(hole-$\lambda 2$), contrary to the use of kind-labels in TCP(hole-F_ω) and TIP(hole-F_ω) in the previous section.

Related work (Scrap type applications [18] and ML^{F} [22, 28])

From the viewpoint of compiler writers, Jay and Peyton Jones [18] introduced implicit System F, called System IF. System IF allows redundant type arguments of $M[A]$ to be implicit such as M with no placeholders, whereas a scrapped argument A can be recovered via matching.

Our principal objective on this work is to find out an essential type annotation that governs (un)decidability of type-related problems. Compared with System IF, hole-application terms still have placeholders [], where our type inference mechanism is based on a decidable fragment of second-order unification. The detailed comparison must be interesting and should be given somewhere for practical application.

Le Botlan and Rémy [22] introduced a type system ML^{F}, by extending ML with full polymorphism as in System F. The language ML^{F} has a family of systems, and Rémy and Yakobowski [28] presented a Church-style version $x\mathrm{ML}^{\mathrm{F}}$ with full type information. As a generalization of a polymorphic type $\forall\alpha.\tau$ of System F, a significant feature of ML^{F} is a flexible quantification $\forall(\alpha \geq \sigma)\tau$, where type variables intuitively range over instances of σ. Accordingly, type abstractions are extended such as $\Lambda(\alpha \geq \sigma)a$. Moreover, as a generalization of type application, $x\mathrm{ML}^{\mathrm{F}}$ uses type instantiation $a\phi$, such that $\Gamma \vdash a\phi : \tau_2$ if $\Gamma \vdash a : \tau_1$ and $\Gamma \vdash \phi : \tau_1 \leq \tau_2$. Here, intuitively the instantiation ϕ transforms the type τ_1 of a into another type τ_2 that is an instance of τ_1. In order to handle instantiation formally, besides typing rules and β-reductions as usual, they introduced type instance rules, type instantiation on types, and reduction rules for terms with instantiations. In this way, ML^{F} established the powerful expressiveness successfully. Although the idea of hole-application is orthogonal, our work also proceeds from the same motivation as theirs using type annotations, still under the traditional framework.

Acknowledgements

We would like to thank deeply all the referees for their careful reading and constructive comments.

References

1 H. P. Barendregt: *The lambda Calculus. Its Syntax and Semantics*, North-Holland, second, revised edition, 1984.

2 H. P. Barendregt: *Lambda calculi with types*, In S. Abramsky, *et al.* editors, *Handbook of Logic in Computer Science*, Vol II, pp. 117–309, Oxford University Press, 1992.

3 H. P. Barendregt, W. Dekkers, R. Statman: *Lambda Calculus with Types*, Cambridge University Press, 2012.

4 G. Barthe, M. H. Sørensen: *Domain-Free Pure Type Systems*, Lecture Notes in Computer Science 1234, pp. 9–20, 1997.

5 H.-J. Boehm: *Partial polymorphic type inference is undecidable*, Proc. IEEE 26th Annual Symposium on Foundations of Computer Science, pp. 339–345, 1985.

6 A. Church: *A formulation of the simple theory of types*, J. Symbolic Logic 5, pp. 56–68, 1940.

7 H. B. Curry: *Functionality in combinatory logic*, Proc. Nat. Acad. Science USA, 20, pp. 584–590, 1934.

8 G. Dowek: *Higher-order unification and matching*, In A. Robinson and A. Voronkov editors, *Handbook of Automated Reasoning*, Elsevier Science Publishers B.V. 2001.

9 K. Fujita: *Domain-free $\lambda\mu$-calculus*, Theoretical Informatics and Applications 34, pp. 433-466, 2000.

10 K. Fujita: *CPS-translation as adjoint*, Theoretical Computer Science 411 (2), pp. 324–340, 2010.

11 K. Fujita, A. Schubert: *Partially typed terms between Church-style and Curry-style*, Lecture Notes in Computer Science 1872, pp. 505-520, 2000.

12 K. Fujita, A. Schubert: *Existential type systems with no types in terms*, Lecture Notes in Computer Science 5608, pp. 112-125, 2009.

13 K. Fujita, A. Schubert: *The undecidability of type related problems in type-free style System* F, Leibniz International Proceedings in Informatics 6, pp. 103–118, 2010.

14 K. Fujita, A. Schubert: *The undecidability of type related problems in the type-free style System* F *with finitely stratified polymorphic types*, Information and Computation 218, pp. 69–87, 2012.

15 W. D. Goldfarb: *The undecidability of the second-order unification problems*, Theoretical Computer Science 13, pp. 225–230, 1981.

16 G. Huet: *The undecidability of unification in third order logic*, Information and Control 22, pp. 257–267, 1973.

17 W. A. Howard: *The formulae-as-types notion of construction*, In J. P. Seldin and J. R. Hindley editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980.

18 B. Jay, S. Peyton Jones: *Scrap your type applications*, Lecture Notes in Computer Science 5133, pp. 2–27, 2008.

19 A. J. Kfoury, J. Tiuryn: *Type Reconstruction in Finite Rank Fragments of the Second-Order* λ*-Calculus*, Information and Computation 98, pp. 228–257, 1992.

20 A. J. Kfoury, J. B. Wells: *A direct algorithm for type inference in the rank-2 fragment of the second-order* λ*-calculus*, Proc. ACM LISP and Functional Programming, pp. 196–207, 1994.

21 D. Leivant: *Polymorphic type inference*, POPL '83: Proc. 10th ACM Symposium on Principles of Programming Languages, pp. 88–98, 1983.

22 D. Le Botlan, D. Rémy: *Recasting* ML^{F}, Information and Computation 207, pp. 726–785, 2009.

23 D. Miller: *A logic programming language with lambda-abstraction, function variables, and simple unification*, J. Logic and Computation 1 (4), pp. 497–536, 1991.

24 K. Nakazawa, M. Tatsuta, Y. Kameyama, H. Nakano: *Type checking and typability in domain-free lambda calculi*, Theoretical Computer Science 412, pp. 6193–6207, 2011.

25 M. Parigot: $\lambda\mu$*-Calculus: An algorithmic interpretation of classical natural deduction*, Lecture Notes in Computer Science 624, pp. 190–201, 1992.

26 F. Pfenning: *Partial polymorphic type inference and higher-order unification*, Proc. ACM Conference on LISP and Functional Programming, pp. 153–163, 1988.

27 F. Pfenning: *On the undecidability of partial polymorphic type reconstruction*, Fundamenta Informaticae 19 (1,2), pp. 185–199, 1993.

28 D. Rémy, B. Yakobowski: *A Church-style intermediate language for* ML^F, Theoretical Computer Science 435, pp. 77–105, 2012.

29 W. Synder, J. H. Gallier: *Higher order unification revisited: Complete sets of transformations*, J. Symbolic Computation 8 (1,2) pp. 101–140, 1989.

30 A. Schubert: *Second-order unification and type inference for Church-style polymorphism*, POPL '98: Proc. 25th ACM Symposium on Principles of Programming Languages, pp. 279–288, 1998.

31 L. S. Van Benthem Jutting: *Typing in Pure Type Systems*, Information and Computation 105, pp. 30–41, 1993.

32 J. B. Wells: *Typability and type checking in system* F *are equivalent and undecidable*, Ann. Pure Appl. Logic 98, pp. 111–156, 1999.

Expressibility in the Lambda Calculus with μ

Clemens Grabmayer[1] **and Jan Rochel**[2]

1 **Department of Philosophy, Utrecht University**
PO Box 80126, 3508 TC Utrecht, The Netherlands
clemens@phil.uu.nl

2 **Department of Information and Computing Sciences**
PO Box 80089, 3508 TB Utrecht
jan@rochel.info

Abstract

We address a problem connected to the unfolding semantics of functional programming languages: give a useful characterization of those infinite λ-terms that are $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressible in the sense that they arise as infinite unfoldings of terms in $\boldsymbol{\lambda}_{\mathsf{letrec}}$, the λ-calculus with letrec. We provide two characterizations, using concepts we introduce for infinite λ-terms: regularity, strong regularity, and binding–capturing chains. It turns out that $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressible infinite λ-terms form a proper subclass of the regular infinite λ-terms. In this paper we establish these characterizations only for expressibility in $\boldsymbol{\lambda}_{\mu}$, the λ-calculus with explicit μ-recursion. We show that for all infinite λ-terms T the following are equivalent: (i): T is $\boldsymbol{\lambda}_{\mu}$-expressible; (ii): T is strongly regular; (iii): T is regular, and it only has finite binding–capturing chains.

We define regularity and strong regularity for infinite λ-terms as two different generalizations of regularity for infinite first-order terms: as the existence of only finitely many subterms that are defined as the reducts of two rewrite systems for decomposing λ-terms. These rewrite systems act on infinite λ-terms furnished with a bracketed prefix of abstractions for collecting decomposed λ-abstractions and keeping the terms closed under decomposition. They differ in which vacuous abstractions in the prefix are removed.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs

Keywords and phrases lambda-calculus, lambda-calculus with letrec, unfolding semantics, regularity for infinite lambda-terms, binding-capturing chain

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.206

1 Introduction

A syntactical core of functional programming languages is formed by $\boldsymbol{\lambda}_{\mathsf{letrec}}$, the λ-calculus with letrec, which can also be viewed as an abstract functional language. Formally, $\boldsymbol{\lambda}_{\mathsf{letrec}}$ is the extension of the λ-calculus by adding the construct letrec for expressing recursion as well as explicit substitution. In a slightly enriched form (of e.g. Haskell's Core language) it is used as an intermediate language for the compilation of functional programs, and as such it is the basis for optimizing program transformations. A calculus that in some respects is weaker than $\boldsymbol{\lambda}_{\mathsf{letrec}}$ is $\boldsymbol{\lambda}_{\mu}$, the λ-calculus with the binding construct μ for μ-recursion. Terms in $\boldsymbol{\lambda}_{\mu}$ can be interpreted directly as terms in $\boldsymbol{\lambda}_{\mathsf{letrec}}$ (expressions $\mu f.M(f)$ as $\mathbf{letrec}\ f = M(f)\ \mathbf{in}\ f$), but translations in the other direction are more complicated, and have weaker properties.

For analyzing the execution behavior of functional programs, and for constructing program transformations, expressions in $\boldsymbol{\lambda}_{\mathsf{letrec}}$ or in $\boldsymbol{\lambda}_{\mu}$ are frequently viewed as finite representations of their unfolding semantics: the infinite λ-term that is obtained by completely unfolding all occurring recursive definitions, the letrec- or μ-bindings, in the expression.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 206–222

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In order to provide a theoretical foundation for such practical tasks, we aim to understand how infinite λ-terms look like that are expressible in $\boldsymbol{\lambda}_{\mathsf{letrec}}$ or in $\boldsymbol{\lambda}_{\mu}$ in the sense that they are infinite unfoldings of expressions from the respective calculus. In particular, we want to obtain useful characterizations of these classes of infinite λ-terms. Quite clearly, any such infinite λ-term must exhibit an, in some sense, repetitive structure that reflects the cyclic dependencies present in the finite description. This is because these dependencies are only 'rolled out', and so are preserved, by a typically infinite, stepwise unfolding process.

For infinite terms over a first-order signature there is a well-known concept of repetitive structure, namely regularity. An infinite term is called 'regular' if it has only a finite number of different subterms. Such infinite terms correspond to trees over ranked alphabets that are regular [4]. Like regular trees also regular terms can be expressed finitely by systems of recursion equations [4], by 'rational expressions' [4, Def. 4.5.3] which correspond to μ-terms (see e.g. [5]), or by terms using letrec-bindings. In this context finite expressions denote infinite terms either via a mathematical definition (a fixed-point construction, or induction on paths) or as the limit of a rewrite sequence consisting of unfolding steps. Regularity of infinite terms coincides, furthermore, with expressibility by finite terms enriched with either of the binding constructs μ or letrec. It is namely well-known that both representations are equally expressive with respect to denoting infinite terms, because a representation using letrec's can also be transformed into one using μ's while preserving the infinite unfolding.

For infinite λ-terms, however, the situation is different: A definition of regularity is less clear due to the presence of variable binding. And there are infinite λ-terms that are regular in an intuitive sense, yet apparently are not $\boldsymbol{\lambda}_{\mathsf{letrec}}$- or $\boldsymbol{\lambda}_{\mu}$-expressible. For example, the syntax trees of the infinite λ-terms T in Fig. 1 and U in Fig. 2 both exhibit a regular structure. But while T clearly is $\boldsymbol{\lambda}_{\mu}$- and $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressible (by $\mu f.\lambda xy.f\,y\,x$ and $\mathbf{letrec}\ f = \lambda xy.f\,y\,x\ \mathbf{in}\ f$, respectively), this seems not to be the case for U: the λ-bindings in U are infinitely entangled, which suggests that it cannot be the result of just an unfolding process. Therefore it appears that the intuitive notion of regularity is too weak for capturing the properties of $\boldsymbol{\lambda}_{\mu}$- and of $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressibility. We note that actually these two properties coincide, because between λ_{μ}-terms and $\lambda_{\mathsf{letrec}}$-terms similar transformations are possible as between representations with μ and with letrec of infinite first-order terms (but this will not be proved here).

It is therefore desirable to obtain a precise, and conceptually satisfying, definition of regularity for infinite λ-terms that formalizes the intuitive notion, and that makes it possible to prove that $\boldsymbol{\lambda}_{\mu}$-/$\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressible infinite λ-terms form only a proper subclass of the regular ones. Furthermore the question arises of whether the property of $\boldsymbol{\lambda}_{\mu}$-/$\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressibility can be captured by a stronger concept of regularity that is still natural in some sense.

We tackle both desiderata at the same time, and provide solutions, but treat only the case of $\boldsymbol{\lambda}_{\mu}$-expressibility here. We introduce two concepts of regularity for infinite λ-terms. For this, we devise two closely related rewrite systems (infinitary Combinatory Reduction Systems) that allow to 'observe' infinite λ-terms by subjecting them to primitive decomposition steps and thereby obtaining 'generated subterms'. Then regular, and strongly regular infinite λ-terms are defined as those that give rise to only a finite number of generated subterms in the respective decomposition system. We establish the inclusion of the class of strongly regular in the class of regular infinite λ-terms, and the fact that this is a proper inclusion (by recognizing that the λ-term U in Fig. 2 is regular, but not strongly regular). As our main result we show that an infinite λ-term is $\boldsymbol{\lambda}_{\mu}$-expressible (that is, expressible by a term in $\boldsymbol{\lambda}_{\mu}$) if and only if it is strongly regular. Here we say that a term M in $\boldsymbol{\lambda}_{\mu}$ expresses an infinite λ-term V if V is the infinite unfolding of M. An infinite unfolding is unique if it exists, and it can be obtained as the limit of an infinite rewrite sequence of unfolding steps.

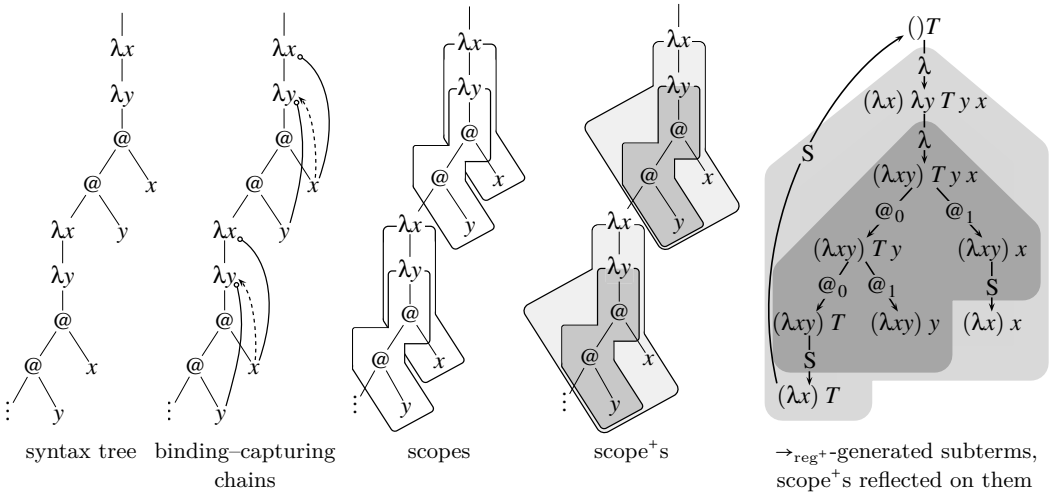


Figure 1 Strongly regular infinite λ-term T, which can be expressed by the λ_μ-term $\mu f.\lambda xy.f\,y\,x$.

This expressibility theorem is a special case of a result we reported in [6], which states that strong regularity coincides with $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressibility. That more general result settles a conjecture by Blom in [3, Sect. 1.2.4]. Its proof is closely connected to the proof of the result on $\boldsymbol{\lambda}_\mu$-expressibility we give here, which exhibits and highlights all the same features, but lacks the complexity that is inherent to the formal treatment of unfolding for terms in $\boldsymbol{\lambda}_{\mathsf{letrec}}$.

Additionally we give a result that explains the relationship between regularity and strong regularity by means of the concept of 'binding–capturing chain': a regular infinite λ-terms is strongly regular if and only if it does not contain an infinite binding–capturing chain.

Overview. In Section 2 we introduce rewriting systems (infinitary CRSs) for decomposing λ-terms into their generated subterms. By means of these systems we define regularity and strong regularity for infinite λ-terms. In Section 3 we provide sound and complete proof systems for these notions. In Section 4 we develop the notion of binding–capturing chain in infinite λ-terms, and show that strong regularity amounts to regularity plus the absence of infinite binding–capturing chains. In Section 5 we establish the correspondence between strong regularity and λ_μ-expressibility for infinite λ-terms. In the final Section 6 we place the results presented here in the context of our investigations about sharing in cyclic λ-terms.

2 Regular and strongly regular infinite λ-terms

In this section we motivate the introduction of higher-order versions of regularity, and subsequently introduce the concepts of regularity and strong regularity for infinite λ-terms.

For higher-order infinite terms such as infinite λ-terms, regularity has been used with as meaning the existence of a first-order syntax tree with named variables that is regular (e.g. in [2, 1]). For example, the infinite λ-terms T and U from Figures 1 and 2 are regular in this sense. However, such a definition of regularity has the drawback that it depends on a first-order representation (as syntax trees with named abstractions and variables) that is not invariant under α-conversion, the renaming of bound variables. Note that the syntax trees of T and U have renaming variants that contain infinitely many variables, and that for this reason are not regular as first-order trees. It is therefore desirable to obtain a definition of regularity that uses the condition for the first-order case but adapts the notion of subterm to λ-terms, and that pertains to a formulation of infinite λ-terms as higher-order terms.

Viable notions of subterm for λ-terms in a higher-order formalization require a stipulation on how to treat variable binding when stepping from a λ-abstraction $\lambda z.V$ into its body V. For this purpose we enrich the syntax of λ-terms with a bracketed prefix of abstractions

(similar to a proof system for weak μ-equality in [5, Fig. 12]), and consider $(\lambda z)V$ as a 'generated subterm' of $\lambda z.V$, obtained by a λ-abstraction decomposition applied to $()\lambda z.V$, where $()$ is the empty prefix. An expression $(\lambda x_1 \dots x_n)T$ represents a partially decomposed λ-term: the body T typically contains free occurrences of variables that in the original λ-term were bound by λ-abstractions but have since been split off by decomposition steps. The role of such abstractions has then been taken over by abstractions in the prefix $(\lambda x_1 \dots x_n)$. In this way expressions with abstraction prefixes are kept closed under decomposition steps.

We formulate infinite λ-terms and their prefixed variants as terms in iCRSs (infinitary Combinatory Reduction Systems) for which we draw on the literature. By *iCRS-terms* we mean α-equivalence classes of iCRS-preterms that are defined by metric completion from finite CRS-terms [10]. For denoting and manipulating infinite terms we use customary notation for finite terms. In order to simplify our exposition we restrict to closed terms, but at one stage (a proof system in Section 5) we allow constants in our terms.

Note that we do not formalize β-reduction since we are only concerned with a static analysis of infinite λ-terms and later with finite expressions that express them via unfolding.

▶ **Definition 1** (iCRS-representation of $\boldsymbol{\lambda^\infty}$)**.** The CRS-signature for the λ-calculus $\boldsymbol{\lambda}$ and the infinitary λ-calculus $\boldsymbol{\lambda^\infty}$ consists of the set $\Sigma_\lambda = \{\mathsf{app}, \mathsf{abs}\}$ where app is a binary and abs a unary function symbol. By $Ter(\boldsymbol{\lambda^\infty})$ we denote the set of infinite closed iCRS-terms over Σ_λ with the restriction that CRS-abstraction can only occur as an argument of an abs-symbol. Note that here and below we subsume finite λ-terms under the infinite ones.

▶ **Definition 2** (iCRS-representation of $\boldsymbol{(\lambda^\infty)}$)**.** The CRS-signature $\Sigma_{(\lambda)}$ for $\boldsymbol{(\lambda^\infty)}$, the version of $\boldsymbol{\lambda^\infty}$ with bracketed abstractions, extends Σ_λ by unary function symbols of arbitrary arity: $\Sigma_{(\lambda)} = \Sigma_\lambda \cup \{\mathsf{pre}_n \mid n \in \mathbb{N}\}$. Prefixed λ-terms $\mathsf{pre}_n([x_1]\dots[x_n]T)$ will informally be denoted by $(\lambda x_1 \dots x_n)T$, abbreviated as $(\lambda \vec{x})T$, or $()T$ in case of an empty prefix. By $Ter(\boldsymbol{(\lambda^\infty)})$ we denote the set of closed iCRS-terms over $\Sigma_{(\lambda)}$ of the form $\mathsf{pre}_n([x_1]\dots[x_n]T)$ for some $n \in \mathbb{N}$ and some term T over the signature Σ_λ with the restriction that a CRS-abstraction can only occur as an argument of an abs-symbol.

▶ **Example 3.** The λ-term $\lambda xy.y\,x$ in CRS-notation is $\mathsf{abs}([x]\mathsf{abs}([y]\mathsf{app}(y,x)))$. The prefixed λ-term $(\lambda x)\lambda y.y\,x$ is represented by $\mathsf{pre}_1([x]\mathsf{abs}([y]\mathsf{app}(y,x)))$.

On these prefixed λ-terms, we define two rewrite strategies $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$ that deconstruct infinite λ-terms by steps that decompose applications and λ-abstractions, and take place just below the marked abstractions. They differ with respect to which vacuous prefix bindings they remove: while $\to_{\mathrm{reg}}$-steps drop such bindings always before steps over applications and λ-abstractions, $\to_{\mathrm{reg}^+}$-steps remove vacuous bindings only if they occur at the end of the abstraction prefix. These rewrite strategies will define respective notions of 'generated subterm', and will give rise to two concepts of regularity: a λ-term is called regular/strongly regular if its set of $\to_{\mathrm{reg}}$-reachable/$\to_{\mathrm{reg}^+}$-reachable generated subterms is finite.

▶ **Definition 4** (decomposing $\boldsymbol{(\lambda^\infty)}$-terms with rewrite strategies $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$)**.** We consider the following CRS-rules over $\Sigma_{(\lambda)}$ in informal notation:[1]

$$
\begin{array}{lll}
(\varrho^{@_i}): & (\lambda x_1 \dots x_n)\,T_0\,T_1 \to (\lambda x_1 \dots x_n)T_i & (i \in \{0,1\}) \\
(\varrho^{\lambda}): & (\lambda x_1 \dots x_n)\,\lambda x_{n+1}.T_0 \to (\lambda x_1 \dots x_{n+1})T_0 & \\
(\varrho^{\mathsf{S}}): & (\lambda x_1 \dots x_{n+1})T_0 \to (\lambda x_1 \dots x_n)T_0 & (\text{if binding } \lambda x_{n+1} \text{ is vacuous}) \\
(\varrho^{\mathrm{del}}): & (\lambda x_1 \dots x_{n+1})T_0 \to (\lambda x_1 \dots x_{i-1} x_{i+1} \dots x_{n+1})T_0 & (\text{if bind. } \lambda x_i \text{ is vacuous})
\end{array}
$$

[1] E.g. explicit form of scheme (ϱ^{S}): $\mathsf{pre}_{n+1}([x_1 \dots x_{n+1}]\,Z(x_1,\dots,x_n)) \to \mathsf{pre}_n([x_1 \dots x_n]\,Z(x_1,\dots,x_n))$.

We call an occurrence o of a binding like a λ-abstraction λz or a CRS-abstraction $[z]$ in a term V *vacuous* if V does not contain a variable occurrence of z that is bound by o.

The iCRS with these rules induces an ARS (abstract rewriting system) $\mathcal{A}$ on infinite terms over $\Sigma_{(\lambda)}$. By (Λ) we denote the sub-ARS of $\mathcal{A}$ with its set of objects restricted to $Ter(\mathbf{(\lambda)})$. Note that $Ter(\mathbf{(\lambda)})$ is closed under steps in (Λ). By $\to_{@_0}$, $\to_{@_1}$, $\to_\lambda$, $\to_{\mathsf{S}}$, $\to_{\mathrm{del}}$ we denote the rewrite relations induced by (Λ)-steps with respect to rules $\varrho^{@_0}$, $\varrho^{@_1}$, ϱ^{λ}, ϱ^{S}, ϱ^{del}. We define *Reg* (*Reg*$^+$) as the sub-ARS of (Λ) that arises from dropping steps that are:

- due to ϱ^{S} (ϱ^{del}), so that the prefix can be shortened only by ϱ^{del}-steps (ϱ^{S}-steps).
- due to rules other than ϱ^{del} (ϱ^{S}) but whose source is also a source of a ϱ^{del}-step (ϱ^{S}-step).

Reg (*Reg*$^+$) is ϱ^{del}-*eager* (ϱ^{S}-eager) in the sense that on each path ϱ^{del}-steps (ϱ^{S}-steps) occur as soon as possible. We denote by $\to_{\mathrm{reg}}$ ($\to_{\mathrm{reg}^+}$) the rewrite strategy induced by *Reg* (*Reg*$^+$).[2]

▶ **Example 5.** Using the recursive equation $T = \lambda xy.T\,y\,x$ as a description for the infinite λ-term T in Fig. 1, we find that decomposition by $\to_{\mathrm{reg}^+}$-steps proceeds as follows, repetitively:

$$()T \quad (\lambda x)\lambda y.T\,y\,x \quad (\lambda xy)T\,y\,x \quad \begin{matrix}(\lambda xy)T\,y \\ \\ (\lambda xy)\,x\end{matrix} \quad \begin{matrix}(\lambda xy)T \\ (\lambda xy)\,y \\ (\lambda x)\,x\end{matrix} \quad (\lambda x)T \quad ()T \quad \dots$$

(in a tree that branches to the right). Note that removal steps for vacuous bindings take place only at the end of the prefix. See Fig. 1 right for the reduction graph of $()T$ with displayed sorts of decomposition steps. Although $\to_{\mathsf{S}}$-steps also are $\to_{\mathrm{del}}$-steps, this decomposition is not also one according to $\to_{\mathrm{reg}}$, because e.g. the step $(\lambda xy)T\,y \to_{@_1} (\lambda xy)\,y$ is not ϱ^{del}-eager.

The rules ϱ^{S} are related to the de Bruijn notation of λ-terms. Consider $\lambda x.(\lambda y.x\,x)\,x$ which in de Bruijn notation is $\lambda.(\lambda.1\,1)\,0$ and when using Peano numerals $\lambda.(\lambda.S(0)\,S(0))\,0$. Now if the symbols S are allowed to appear 'shared' and occur further up in the term as in $\lambda.(\lambda.S(0\,0))\,0$, then this term structure corresponds to the decomposition with $\to_{\mathrm{reg}^+}$.

To understand the difference between $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$, consider the notions of scope and scope$^+$, illustrated in Figures 1 and 2. The scope of an abstraction is the smallest connected portion of a syntax tree that contains the abstraction itself as well as all of its bound variable occurrences. And scope$^+$s extend scopes minimally so that the resulting areas appear properly nested. For a precise definition we refer to [6, Sect. 4]. As can be seen in Figures 1 and 2, applications of ϱ^{del} (ϱ^{S}) coincide with the positions where scopes (scope$^+$s) are closed.

▶ **Definition 6** (regular/strongly regular λ-terms, generated subterms)**.** Let $T \in Ter(\boldsymbol{\lambda^\infty})$. We define the sets $ST(T)$ and $ST^+(T)$ of *generated subterms* of T *with respect to* $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$:

$$ST(T) := \{U \in Ter(\mathbf{(\lambda^\infty)}) \mid ()T \twoheadrightarrow_{\mathrm{reg}} U\} \qquad ST^+(T) := \{U \in Ter(\mathbf{(\lambda^\infty)}) \mid ()T \twoheadrightarrow_{\mathrm{reg}^+} U\}$$

We say that T is *regular* (*strongly regular*) if T has only finitely many generated subterms with respect to $\to_{\mathrm{reg}}$ (respectively, with respect to $\to_{\mathrm{reg}^+}$).

▶ **Example 7.** From the $\to_{\mathrm{reg}^+}$-decomposition in Example 5 and Fig. 1 of the infinite λ-term T in Fig. 1 it follows that $ST^+(T)$ consists of 9 generated subterms. Hence T is strongly regular.

The situation is different for the infinite λ-term U in Fig. 2. When represented as the term $\lambda x.R(x)$ together with the CRS-rule $R(X) \to \lambda y.R(y)\,X$, its $\to_{\mathrm{reg}^+}$-decomposition is:

$$()U \quad (\lambda x)\,R(x) \quad (\lambda xy)\,R(y)\,x \quad \begin{matrix}(\lambda xy)\,R(y) \\ (\lambda xy)\,x\end{matrix} \quad \begin{matrix}(\lambda xyz)\,R(z)\,y \\ (\lambda x)\,x\end{matrix} \quad \begin{matrix}(\lambda xyz)\,R(z) \\ (\lambda xyx)\,y\end{matrix} \quad \begin{matrix}(\lambda xyzu)\,R(u)\,z \\ (\lambda xy)\,y\end{matrix} \quad \dots$$

[2] We use 'rewrite strategy' for a relation on terms, and not for a sub-ARS of a CRS-induced ARS [13].

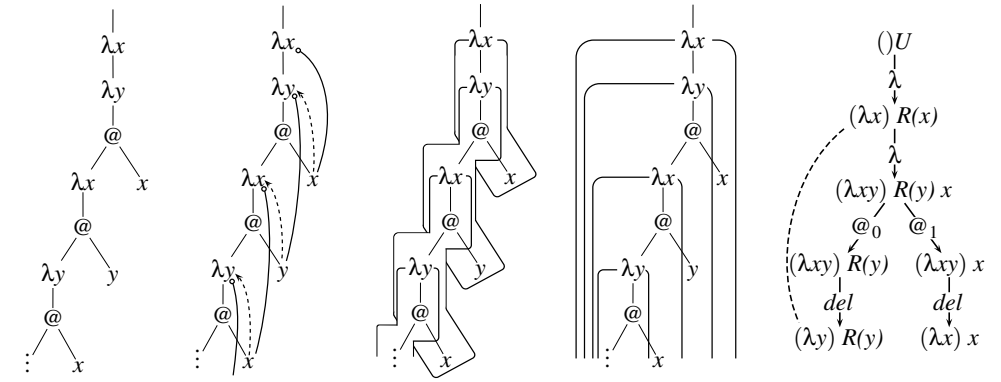


Figure 2 The regular infinite λ-term U that is not strongly regular, and not λ_μ-expressible.

Since here the prefixes grow unboundedly, U has infinitely many $\to_{\mathrm{reg}^+}$-generated subterms, and hence U is not strongly regular. But its $\to_{\mathrm{reg}}$-decomposition exhibits again a repetition as can be seen from the reduction graph in Fig. 2 on the right. Note that a vacuous binding from within a prefix is removed. $()U$ has 6 only different $\to_{\mathrm{reg}}$-reducts. Hence U is regular.

For infinite λ-terms like $(\lambda x_1.x_1)\,(\lambda x_1.\lambda x_2.x_2)\,(\lambda x_1.\lambda x_2.\lambda x_3.x_3)\ldots$ that do not have any regular pseudoterm syntax-trees, both $\to_{\mathrm{reg}^+}$-decomposition and $\to_{\mathrm{reg}}$-decomposition yield infinitely many generated subterms, and hence they are neither regular nor strongly regular.

For a better understanding of the precise relationship between $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$, and eventually of the two concepts of generated subterm and of regularity, we gather a number of basic properties of these rewrite strategies and their constituents.

▶ **Proposition 8.** The restrictions of the rewrite relations from Def. 4 to $Ter(\boldsymbol{(\lambda^\infty)})$, the set of objects of *Reg* and Reg^+, have the following properties:

(i) $\to_{\mathrm{del}}$ is confluent, and terminating.
(ii) $\to_{\mathsf{S}} \subseteq \to_{\mathrm{del}}$. Furthermore, $\to_{\mathsf{S}}$ is deterministic, hence confluent, and terminating.
(iii) $\to_{\mathrm{del}}$ one-step commutes with $\to_\lambda$, $\to_{@_0}$, $\to_{@_1}$, and one-step sub-commutes with $\to_{\mathsf{S}}$; $\to_{\mathrm{del}}$ postpones over $\to_\lambda$, $\to_{@_0}$, $\to_{@_1}$ and $\to_{\mathsf{S}}$. Formulated symbolically, this means:

$$\leftarrow_{\mathrm{del}} \cdot \to_\lambda \subseteq \to_\lambda \cdot \leftarrow_{\mathrm{del}} \qquad \leftarrow_{\mathrm{del}} \cdot \to_{@_i} \subseteq \to_{@_i} \cdot \leftarrow_{\mathrm{del}} \qquad \leftarrow_{\mathrm{del}} \cdot \to_{\mathsf{S}} \subseteq \to_{\mathsf{S}}^{=} \cdot \leftarrow_{\mathrm{del}}^{=}$$
$$\to_{\mathrm{del}} \cdot \to_\lambda \subseteq \to_\lambda \cdot \to_{\mathrm{del}} \qquad \to_{\mathrm{del}} \cdot \to_{@_i} \subseteq \to_{@_i} \cdot \to_{\mathrm{del}} \qquad \to_{\mathrm{del}} \cdot \to_{\mathsf{S}} \subseteq \to_{\mathsf{S}} \cdot \to_{\mathrm{del}}$$

(iv) Normal forms of $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$ are of the form $(\lambda x)\,x$, and $(\lambda x_1 \ldots x_n)\,x_n$, respectively.
(v) $\to_{\mathrm{reg}}$ and $\to_{\mathrm{reg}^+}$ are finitely branching, and, on finite terms, terminating.

Proof. These properties, including those concerning commutation of steps, are easy to verify by analyzing the behavior of the rewrite rules in *Reg* on terms of $Ter(\boldsymbol{(\lambda^\infty)})$. ◀

▶ **Proposition 9.** (i) Let $(\lambda\vec{x})T$ be a term in $Ter(\boldsymbol{(\lambda^\infty)})$ with $|\vec{x}| = n \in \mathbb{N}$. Then the number of terms $(\lambda\vec{y})U$ in $Ter(\boldsymbol{(\lambda^\infty)})$ with $(\lambda\vec{y})U \twoheadrightarrow_{\mathrm{del}} (\lambda\vec{x})T$ and $|\vec{y}| = n + k \in \mathbb{N}$ is $\binom{n+k}{n}$.
(ii) Let $A \subseteq Ter(\boldsymbol{(\lambda^\infty)})$ be a finite set, and $k \in \mathbb{N}$. Then also the set of terms in $Ter(\boldsymbol{(\lambda^\infty)})$ that are the form $(\lambda\vec{y})U$ with $|\vec{y}| \le k$ and that have a $\twoheadrightarrow_{\mathrm{del}}$-reduct in A is finite.

We state a lemma about a close connection between $\to_{\mathrm{reg}}$- and $\to_{\mathrm{reg}^+}$-rewrite sequences.

▶ **Lemma 10.** (i) *On* $Ter(\boldsymbol{\lambda^\infty})$ *it holds:* $\twoheadleftarrow_{\mathrm{del}} \cdot \to_{\mathrm{reg}^+} \subseteq \to_{\mathrm{del}}^{!} \cdot \to_{\mathrm{reg}}^{=} \cdot \twoheadleftarrow_{\mathrm{del}}$, *where* $\to_{\mathrm{del}}^{!}$ *denotes many-step* $\to_{\mathrm{del}}$*-reduction to* $\to_{\mathrm{del}}$*-normal form. As a consequence of this and of* $\to_{\mathrm{del}}^{!} \cdot \to_{\mathrm{reg}}^{=} \subseteq \twoheadrightarrow_{\mathrm{reg}}$, *every finite or infinite rewrite sequence in* $Ter(\boldsymbol{\lambda^\infty})$ *of the form:*

$$\tau : (\lambda\vec{x}_0)T_0 \to_{\mathrm{reg}^+} (\lambda\vec{x}_1)T_1 \to_{\mathrm{reg}^+} \ldots \to_{\mathrm{reg}^+} (\lambda\vec{x}_k)T_k \to_{\mathrm{reg}^+} \ldots$$

projects over a sequence $\pi : (\lambda\vec{x}_0)T_0 \twoheadrightarrow_{\mathrm{del}} (\lambda\vec{x}'_0)T_0$ *to a rewrite sequence of the form:*

$$\check{\tau} : (\lambda\vec{x}'_0)T_0 \twoheadrightarrow_{\mathrm{reg}} (\lambda\vec{x}'_1)T_1 \twoheadrightarrow_{\mathrm{reg}} \ \ldots \ \twoheadrightarrow_{\mathrm{reg}} (\lambda\vec{x}'_k)T_k \twoheadrightarrow_{\mathrm{reg}} \ldots$$

in the sense that $(\lambda\vec{x}_k)T_k \twoheadrightarrow_{\mathrm{del}} (\lambda\vec{x}'_k)T_k$ *for all* $k \in \mathbb{N}$ *less or equal to the length of* τ.

(ii) *On* $Ter(\boldsymbol{\lambda^\infty})$ *it holds:* $\twoheadrightarrow_{\mathrm{del}} \cdot \to_{\mathrm{reg}} \subseteq \to_{\mathsf{S}}^{!} \cdot \to_{\mathrm{reg}^+}^{=} \cdot \twoheadrightarrow_{\mathrm{del}}$. *Due to this and* $\to_{\mathsf{S}}^{!} \cdot \to_{\mathrm{reg}^+}^{=} \subseteq \twoheadrightarrow_{\mathrm{reg}^+}$, *every rewrite sequence* $\tau : (\lambda\vec{x}'_0)T_0 \to_{\mathrm{reg}} (\lambda\vec{x}'_1)T_1 \to_{\mathrm{reg}} \ldots \to_{\mathrm{reg}} (\lambda\vec{x}'_k)T_k \to_{\mathrm{reg}} \ldots$ *in* $Ter(\boldsymbol{\lambda^\infty})$ *lifts over a sequence* $\pi : (\lambda\vec{x}_0)T_0 \twoheadrightarrow_{\mathrm{del}} (\lambda\vec{x}'_0)T_0$ *to a* $\to_{\mathrm{reg}^+}$*-rewrite sequence of the form:* $\hat{\tau} : (\lambda\vec{x}_0)T_0 \twoheadrightarrow_{\mathrm{reg}^+} (\lambda\vec{x}_1)T_1 \twoheadrightarrow_{\mathrm{reg}^+} \ \ldots \ \twoheadrightarrow_{\mathrm{reg}^+} (\lambda\vec{x}_k)T_k \twoheadrightarrow_{\mathrm{reg}^+} \ldots$ *in the sense that* $(\lambda\vec{x}_k)T_k \twoheadrightarrow_{\mathrm{del}} (\lambda\vec{x}'_k)T_k$ *for all* $k \in \mathbb{N}$ *less or equal to the length of* τ.

Proof. The inclusion properties in (10) and (10) can be shown by easy arguments with diagrams using the commutation properties in Prop. 8, (iii), as well as (i) and (ii) from there. ◀

Now we are able to establish that strong regularity implies regularity for infinite λ-terms.

▶ **Proposition 11.** Every strongly regular infinite λ-term is also regular. Finite λ-terms are both regular and strongly regular.

Proof. Let T be a strongly regular infinite λ-term. Therefore $ST^+(T)$ is finite. Since every $\to_{\mathrm{reg}}$-rewrite-sequence from $()T$ lifts to a $\to_{\mathrm{reg}^+}$-rewrite-sequence from $()T$ over $\twoheadrightarrow_{\mathrm{del}}$-compression due to Lemma 10, (10), every term in $ST(T)$ is the $\twoheadrightarrow_{\mathrm{del}}$–compression of a term in $ST^+(T)$. Then it follows by Prop. 9, (i), that also $ST(T)$ is finite. Hence T is also regular.

Let T be a finite λ-term. Due to to Prop. 8, (v), Kőnig's Lemma can be applied to the reduction graph of $()T$ with respect to $\twoheadrightarrow_{\mathrm{reg}^+}$ to yield that T has only finitely many generated subterms with respect to $\twoheadrightarrow_{\mathrm{reg}^+}$. Hence T is strongly regular. ◀

3 Proving regularity and strong regularity

In this section we introduce proof systems for regularity and strong regularity: the systems $\mathbf{Reg}^{\boldsymbol{\infty}}$ and $\mathbf{Reg}^{\boldsymbol{+,\infty}}$ with typically infinite derivations, and the systems $\mathbf{Reg}$, $\mathbf{Reg}^{\boldsymbol{+}}$, and $\mathbf{Reg}^{\boldsymbol{+}}_{\mathbf{0}}$ for provability by finite derivations. A completed derivation of $()U$ in $\mathbf{Reg}^{\boldsymbol{\infty}}$ (in $\mathbf{Reg}^{\boldsymbol{+,\infty}}$) corresponds to the 'tree unfolding' of the $\to_{\mathrm{reg}}$-reduction graph (the $\to_{\mathrm{reg}^+}$-reduction graph) of $()U$, which is a tree that describes all $\to_{\mathrm{reg}}$-(resp. $\to_{\mathrm{reg}^+}$-)rewrite sequences from $()U$. Closed derivations of $()U$ in $\mathbf{Reg}$ (in $\mathbf{Reg}^{\boldsymbol{+}}$, or $\mathbf{Reg}^{\boldsymbol{+}}_{\mathbf{0}}$) correspond to finite unfoldings of the $\to_{\mathrm{reg}}$-reduction graph (the $\to_{\mathrm{reg}^+}$-reduction graph) into a graph with only vertical sharing.

We start by introducing proof systems for well-formed prefixed terms (terms in $Ter(\boldsymbol{(\lambda^\infty)})$).

▶ **Definition 12** (proof systems $\boldsymbol{(\Lambda)^\infty}$, $\boldsymbol{(\Lambda)^{+,\infty}}$ for well-formed $\boldsymbol{\lambda^\infty}$-terms)**.** The proof systems defined here act on CRS-terms over signature $\Sigma_{(\lambda)}$ as formulas, and are Hilbert-style systems for potentially infinite prooftrees (of depth $\leq \omega$). The system $\boldsymbol{(\Lambda)^{+,\infty}}$ has the axioms (0) and the rules (@), (λ), and (S) in Fig. 3. The system $\boldsymbol{(\Lambda)^\infty}$ arises from $\boldsymbol{(\Lambda)^{+,\infty}}$ by replacing the axioms (0) and the rule (S) with the axioms (0) and the rule (del) in Fig. 4, respectively.

A finite or infinite derivation $\mathcal{T}$ in $\boldsymbol{(\Lambda)^\infty}$ (in $\boldsymbol{(\Lambda)^{+,\infty}}$) is called *closed* if all terms in leafs of $\mathcal{T}$ are axioms. Derivability of a term $(\lambda\vec{x})T$ in $\boldsymbol{(\Lambda)^\infty}$ (in $\boldsymbol{(\Lambda)^{+,\infty}}$), symbolically $\vdash_{\boldsymbol{(\Lambda)^\infty}} (\lambda\vec{x})T$ (resp. $\vdash_{\boldsymbol{(\Lambda)^{+,\infty}}} (\lambda\vec{x})T$), means the existence of a closed derivation with conclusion $(\lambda\vec{x})T$.

▶ **Definition 13** (proof systems $\mathbf{Reg}^{\boldsymbol{\infty}}$, $\mathbf{Reg}^{\boldsymbol{+,\infty}}$)**.** The proof systems $\mathbf{Reg}^{\boldsymbol{\infty}}$ and $\mathbf{Reg}^{\boldsymbol{+,\infty}}$ have the same axioms and rules as $\boldsymbol{(\Lambda)^\infty}$ and $\boldsymbol{(\Lambda)^{+,\infty}}$, respectively, but they restrict the notion of derivability. A derivation $\mathcal{D}$ in $\boldsymbol{(\Lambda)^\infty}$ (in $\boldsymbol{(\Lambda)^{+,\infty}}$) is called *admissible in* $\mathbf{Reg}^{\boldsymbol{\infty}}$ (*in* $\mathbf{Reg}^{\boldsymbol{+,\infty}}$)

$$\frac{}{(\lambda\vec{x}y)\,y}\;\mathsf{0} \qquad \frac{(\lambda\vec{x}y)\,T_0}{(\lambda\vec{x})\,\lambda y.T_0}\;\lambda \qquad \frac{(\lambda\vec{x})\,T_0 \qquad (\lambda\vec{x})\,T_1}{(\lambda\vec{x})\,T_0\,T_1}\;@$$

$$\frac{(\lambda x_1 \dots x_{n-1})\,T}{(\lambda x_1 \dots x_n)\,T}\;\mathsf{S} \quad \text{(if the binding } \lambda x_n \text{ is vacuous)} \qquad \frac{\begin{array}{c}[(\lambda\vec{x})\,T]^l \\ \mathcal{D}_0\end{array}}{(\lambda\vec{x})\,T}\;\mathrm{FIX}, l \quad (\text{if } |\mathcal{D}_0| \geq 1)$$

Figure 3 The proof system $\mathbf{Reg}^+$ for strongly regular λ-terms. In the variant system $\mathbf{Reg}^+_{\mathbf{0}}$ of $\mathbf{Reg}^+$, instances of (FIX) are subject to the additional side-condition: for all $(\lambda\vec{y})\,U$ on threads in $\mathcal{D}_0$ from open marked assumptions $((\lambda\vec{x})\,T)^u$ downwards it holds that $|\vec{y}| \geq |\vec{x}|$. The systems $\mathbf{(\Lambda)}^{+,\infty}$ and $\mathbf{Reg}^{+,\infty}$ do not contain the rule FIX. Derivations in $\mathbf{Reg}^+$, $\mathbf{Reg}^+_{\mathbf{0}}$, and $\mathbf{Reg}^\infty$ must be (S)-eager.

$$\frac{}{(\lambda y)\,y}\;\mathsf{0} \qquad \frac{(\lambda x_1 \dots x_{i-1}x_{i+1} \dots x_n)\,T}{(\lambda x_1 \dots x_n)\,T}\;\mathrm{del} \quad \text{(if the binding } \lambda x_i \text{ is vacuous)}$$

Figure 4 The proof system $\mathbf{Reg}$ for regular λ-terms arises from $\mathbf{Reg}^+$ through replacing the rule (S) by the rule (del), and the axiom scheme (0) by the more restricted version here. The systems $\mathbf{(\Lambda)}^\infty$ and $\mathbf{Reg}^\infty$ do not contain the rule (FIX). Derivations in $\mathbf{Reg}$ and $\mathbf{Reg}^\infty$ must be (del)-eager.

if it contains only finitely many different terms, and if it is (del)-*eager* ((S)-*eager*), that is, if no conclusion of an instance of (@) or (λ) in $\mathcal{D}$ is the source of a $\to_{\mathrm{del}}$-step (a $\to_{\mathsf{S}}$-step). Derivability in $\mathbf{Reg}^\infty$ (in $\mathbf{Reg}^{+,\infty}$) means the existence of a closed admissible derivation.

We say that a proof system $\mathcal{S}$ is *sound* (*complete*) for a property P of infinite λ-terms if $\vdash_{\mathcal{S}} ()T$ implies $P(T)$ (if $P(T)$ implies $\vdash_{\mathcal{S}} ()T$) for all infinite λ-terms T.

The systems $\mathbf{(\Lambda)}^\infty$ and $\mathbf{(\Lambda)}^{+,\infty}$ are sound and complete for all infinite λ-terms in $Ter(\boldsymbol{\lambda}^\infty)$: for completeness note that every prefixed term $(\lambda\vec{y})\,U$ with U not a variable is the conclusion of an instance of a rule in these systems. This leads us to statements for $\mathbf{Reg}^\infty$ and $\mathbf{Reg}^{+,\infty}$.

▶ **Proposition 14.** (i) $\mathbf{Reg}^\infty$ is sound and complete for regularity of infinite λ-terms.
(ii) $\mathbf{Reg}^{+,\infty}$ is sound and complete for strong regularity of infinite λ-terms.

Proof. We argue only for (ii), since (i) can be seen analogously. Every (S)-eager derivation $\mathcal{T}$ in $\mathbf{(\Lambda)}^{+,\infty}$ with conclusion $()T$ assembles the maximal $\to_{\mathrm{reg}^+}$-rewrite sequences from $()T$ in the following sense: the steps of every such rewrite sequence correspond to the steps through $\mathcal{T}$ along a thread from the conclusion upwards. Therefore if $\mathcal{T}$ is an admissible derivation in $\mathbf{Reg}^{+,\infty}$, and hence contains only finitely many terms, then $ST^+(T)$ is finite. Since every term $()T$ in $Ter(\mathbf{(\lambda)})$ has a (S)-eager derivation in $\mathbf{(\Lambda)}^{+,\infty}$, the converse holds as well. ◀

▶ **Definition 15** (proof systems $\mathbf{Reg}$, $\mathbf{Reg}^+$, and $\mathbf{Reg}^+_{\mathbf{0}}$)**.** The natural-deduction style proof system $\mathbf{Reg}^+$ has the axioms and rules in Fig. 3. Its variant $\mathbf{Reg}^+_{\mathbf{0}}$ demands an additional side-condition on instances of the rule (FIX) as described there. The system $\mathbf{Reg}$ arises from $\mathbf{Reg}^+$ by dropping the rule (S), and restricting the axioms to the axioms (0) in Fig. 4.

A derivation in one of these systems is called *closed* if it does not contain any undischarged marker assumptions (discharging assumptions is indicated by assigning the appertaining assumption markers to instances of FIX, see Fig. 3). Derivability in $\mathbf{Reg}$ (in $\mathbf{Reg}^+$ or in $\mathbf{Reg}^+_{\mathbf{0}}$) means the existence of a closed, (del)-eager ((S)-eager), finite derivation.

The proposition below explains that the side-condition '$|\mathcal{D}_0| \geq 1$' on subderivations of FIX-instances guarantees a 'guardedness' property for threads in derivations in these systems.

▶ **Proposition 16.** Let $\mathcal{D}$ be a derivation in **Reg**, **Reg**$^+$, or **Reg**$^+_\mathbf{0}$. Then for every instance ι of the rule (FIX) in $\mathcal{D}$ it holds: every thread from ι upwards to a marked assumption that is discharged at ι passes at least one instance of a rule (λ) or (@).

Proof. Let $\mathcal{D}$ be a derivation in **Reg**, as the argument is analogous for **Reg**$^+$ and **Reg**$^+_\mathbf{0}$. Let ι be an instance of (FIX) in $\mathcal{D}$, and π a thread from the conclusion $(\lambda\vec{y})U$ of ι to a marked assumption $((\lambda\vec{y})U)^l$ that is discharged at ι. Then due to the side-condition on the topmost instance κ of (FIX) passed on π there is at least one instance of a rule (λ), (@), or (del) passed on π above κ. We are done unless that is an instance of (del). But then there must also be an instance of (λ) on π, since (del) decreases the prefix length, only (λ) increases it, and the prefix lengths in the formula at the start and at the end of π are the same. ◀

▶ **Example 17.** (i) The following are two derivations in **Reg**$^+$ of different efficiency of the infinite λ-term T from Fig. 1 when represented by the recursive equation $T = \lambda xy.T\,y\,x$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{((\,)T)^l}{(\lambda x)T}\,\mathsf{S}}{(\lambda xy)T}\,\mathsf{S} \quad \dfrac{}{(\lambda xy)y}\,0}{(\lambda xy)T\,y}\,@ \quad \dfrac{}{(\lambda x)x}\,0}{(\lambda xy)x}\,\mathsf{S}}{(\lambda xy)T\,y\,x}\,@}{(\lambda x)\,\lambda y.T\,y\,x}\,\lambda}{(\,)\lambda xy.T\,y\,x}\,\lambda}{(\,)T}\,\mathrm{FIX},l \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{((\lambda x)\,\lambda y.T\,y\,x)^l}{(\,)T}\,\lambda}{(\lambda x)T}\,\mathsf{S}}{(\lambda xy)T}\,\mathsf{S} \quad \dfrac{}{(\lambda xy)y}\,0}{(\lambda xy)T\,y}\,@ \quad \dfrac{}{(\lambda x)x}\,0}{(\lambda xy)x}\,\mathsf{S}}{(\lambda xy)T\,y\,x}\,@}{(\lambda x)\,\lambda y.T\,y\,x}\,\lambda}{(\lambda x)\,\lambda y.T\,y\,x}\,\mathrm{FIX},l}{(\,)T}\,\lambda$$

Note that only the left derivation is one in **Reg**$^+_\mathbf{0}$, because the right one contains a term with shorter prefix than the discharged assumption on a thread to the instance of FIX.

(ii) The infinite λ-term from Fig. 2, denoted by the term $(\,)\lambda x.R(x)$ and generated by the CRS-rule $R(X) \to \lambda x.R(x)\,X$ is derivable in **Reg** by the closed derivation on the left, but it is not derivable in **Reg**$^+$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overbrace{((\lambda y)\,R(y))^l}^{=(\lambda x)\,R(x)}}{(\lambda xy)\,R(y)}\,\mathrm{del} \quad \dfrac{}{(\lambda x)x}\,0}{(\lambda xy)\,R(y)\,x}\,\mathrm{del}\,@}{(\lambda x)\,\lambda y.R(y)\,x}\,\lambda}{(\lambda x)\,R(x)}\,\mathrm{FIX},\,l}{(\,)\lambda x.R(x)}\,\lambda$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{(\lambda xyzuv)\,R(v)\,u}\,\lambda \quad \dfrac{}{(\lambda xyz)z}\,0}{(\lambda xyzu)\,\lambda v.R(v)\,u}\,\mathsf{S}}{(\lambda xyzu)\,R(u)\,z}\,@ \quad \dfrac{}{(\lambda xy)y}\,0}{(\lambda xyz)\,\lambda u.R(u)\,z}\,\lambda}{(\lambda xyz)\,R(z)\,y}\,@ \quad \dfrac{}{(\lambda x)x}\,0}{(\lambda xy)\,\lambda z.R(z)\,y}\,\lambda}{(\lambda xy)\,R(y)\,x}\,\mathsf{S}\,@}{(\lambda x)\,\lambda y.R(y)\,x}\,\lambda}{(\,)\lambda x.R(x)}\,\lambda$$

The latter follows from the infinite prooftree on the right, the result of a bottom-up proof search in **Reg**$^+$, which is a derivation in $\mathbf{(\Lambda)^{+,\infty}}$ but not in **Reg**$^{+,\infty}$, since, as it does not contain repetitions, the rule FIX cannot be used to cut off repetitive subderivations.

Finally, we can link derivability in **Reg** and **Reg**$^+$ to regularity and strong regularity.

▶ **Theorem 18.** (i) **Reg** *is sound and complete for regularity of infinite λ-terms.*
(ii) **Reg**$^+$ *and* **Reg**$^+_\mathbf{0}$ *are sound and complete for strong regularity of infinite λ-terms.*

Proof. For (18), in view of Prop. 14, (i), it suffices to be able to transform closed, admissible derivations in **Reg**$^\infty$ into closed derivations in **Reg**, and vice versa. Every closed derivation $\mathcal{D}$ in **Reg** can be unfolded by a stepwise, typically infinite process into a closed derivation in $\mathbf{(\Lambda)^\infty}$: in every step the subderivation of a bottommost instance ι of FIX is transferred to

above each of the marked assumptions that are discharged at ι, and the original instance of FIX is removed. If this process is infinite, then due to Prop. 16 it always eventually increases the size of the part of the derivation below the bottommost occurrences of FIX. Hence in the limit it produces a closed, (del)-eager derivation in $\mathbf{(\Lambda)^\infty}$ that contains only finitely many terms (only those in $\mathcal{D}$), and thus is admissible in $\mathbf{Reg^\infty}$. Conversely, every admissible, closed derivation $\mathcal{T}$ in $\mathbf{Reg^\infty}$ can be 'folded' into a finite closed derivation in $\mathbf{Reg}$ by introducing FIX-instances to cut off the derivation above the upper occurrence of a repetition. This yields a finite derivation since due to admissibility of $\mathcal{T}$ in $\mathbf{Reg^\infty}$ every sufficiently long thread contains a repetition, and then Kőnig's Lemma can be applied.

For $\mathbf{Reg^+}$ in (18) it can be argued analogously, using Prop. 14, (ii), and unfolding/folding between closed derivations in $\mathbf{Reg^+}$ and closed, admissible derivations in $\mathbf{Reg^{+,\infty}}$. Soundness of $\mathbf{Reg_0^+}$ follows from soundness of $\mathbf{Reg^+}$. For completeness of $\mathbf{Reg_0^+}$, note that every closed, admissible derivation $\mathcal{T}$ in $\mathbf{Reg^{+,\infty}}$ can be 'folded' into a closed derivation of $\mathbf{Reg_0^+}$ by using a stricter version of repetition of terms: distinct occurrences of a term $(\lambda\vec{y})U$ on a thread of a prooftree form such a repetition only if all formulas in between have an equally long or longer abstraction prefix. Since $\mathcal{T}$ is admissible, on every infinite thread θ of $\mathcal{T}$ there must occur such a stricter form of repetition, namely of a term with the shortest abstraction prefix among the terms that occur infinitely often on θ. ◀

4 Binding–Capturing Chains

In this section we develop a characterization of strongly regular infinite λ-terms through a property of their term structure, concerning 'binding–capturing chains' on positions of the term. While not needed for obtaining the result concerning $\boldsymbol{\lambda}_{\mathsf{letrec}}$-expressibility in Section 5, we think that this characterization is of independent interest. However, we only outline its proof here, which can be found in [6] and in a report [7] that accompanies this article.

Binding–capturing chains originate from the notion of 'gripping' due to Melliès [11], and from techniques concerning the notion of 'holding' of redexes developed by van Oostrom [12]. In [5] they have been used to study α-conversion-avoiding μ-unfolding.

Technically, binding–capturing chains are alternations of two kinds of links between positions of variable occurrences and λ-abstractions (called binders below) in a λ-term: 'binding links' from a λ-abstraction downward to the variable occurrences it binds, and 'capturing links' from a variable occurrence upward to λ-abstractions that do not bind it, but are situated on the upward path to its binding λ-abstraction. We formalize these links by binding and capturing relations, which are then used to define binding–capturing chains.

▶ **Definition 19** (binding, capturing)**.** For every $T \in Ter(\boldsymbol{\lambda}^\infty)$ we define two binary relations on the set $Pos(T)$ of positions of T: the *binding relation* $\multimap$, and the *capturing relation* $\dashrightarrow$. (For positions in iCRS-terms, see [10].) For defining these relations let $p, q \in Pos(T)$.

$p \multimap q$ (in words: a binder, that is, a λ-abstraction, at p *binds* a variable occurrence at q) holds if p is a binder position, and q a variable position in T, and the binder at position p binds the variable occurrence at position q.

$q \dashrightarrow p$ (in words: a variable occurrence at q *is captured by* a binder at p), and conversely $p \dashleftarrow q$ (the binder at p *captures* a variable occurrence at q), hold if q is a variable position and $p < q$ a binder position in T, and there is no binder position q_0 in T with $p \leq q_0$ and $q_0 \multimap q$.

▶ **Definition 20** (binding–capturing chain)**.** Let $T \in Ter(\boldsymbol{\lambda}^\infty)$. A finite or infinite sequence $\langle p_0, q_1, p_1, q_2, p_2, \ldots\rangle$ in $Pos(T)$ is called a *binding–capturing chain in* T if it links positions alternatingly via binding and capturing: $p_1 \multimap q_2 \dashrightarrow p_2 \multimap q_3 \dashrightarrow p_3 \multimap \ldots$, starting with a binding and ending with a capturing. Its *length* is the number of 'is captured by' links.

See Figs. 1 and 2 for illustrations of binding–capturing chains in terms we have encountered. Next we introduce a position-annotated variant $Reg^{+}_{\boldsymbol{pos}}$ of Reg^{+} in order to relate binding–capturing chains to rewrite sequences in Reg^{+}. The idea is that if a λ-term T has a generated subterm $(\lambda y_1 \ldots y_n)U$ in Reg^{+}, then $(\lambda y_1 \ldots y_n)^{q}_{p_1,\ldots,p_n}U$ is a generated subterm in $Reg^{+}_{\boldsymbol{pos}}$, where $p_1,\ldots,p_n$ are the positions in T from which the bindings $\lambda y_1 \ldots y_n$ in the abstraction prefix descend, and q is the position in T of the body U of this generated subterm.

▶ **Definition 21** (position-annotated variant $Reg^{+}_{\boldsymbol{pos}}$ of Reg^{+})**.** On $Ter(\boldsymbol{(\lambda^{\infty})})$ we consider the following rewrite rules in informal notation:

$$(\varrho^{@_i}_{pos}):\ (\lambda x_1 \ldots x_n)^{q}_{p_1,\ldots,p_n} T_0\, T_1 \to (\lambda x_1 \ldots x_n)^{qi}_{p_1,\ldots,p_n} T_i \qquad \text{(for each } i \in \{0,1\})$$
$$(\varrho^{\lambda}_{pos}):\ (\lambda x_1 \ldots x_n)^{q}_{p_1,\ldots,p_n} \lambda y.T_0 \to (\lambda x_1 \ldots x_n y)^{q00}_{p_1,\ldots,p_n,q} T_0$$
$$(\varrho^{\mathsf{S}}_{pos}):\ (\lambda x_1 \ldots x_{n+1})^{q}_{p_1,\ldots,p_{n+1}} T_0 \to (\lambda x_1 \ldots x_n)^{q}_{p_1,\ldots,p_n} T_0 \qquad \text{(if binding } \lambda x_{n+1} \text{ is vacuous)}$$

The change of the term-body position in a λ-decomposition step is motivated by the underlying CRS-notation for terms in $\boldsymbol{(\lambda^{\infty})}$: when a term $\mathsf{abs}([y]T_0)$ representing a λ-abstraction starts at position q, its binding is declared at position $q0$, and its body T_0 starts at position $q00$.

By Reg^{+}_{pos} we denote the abstract rewriting system induced, similar to the definition of Reg^{+} in Def. 4 earlier, by the rules above on position-annotated terms in $Ter(\boldsymbol{(\lambda^{\infty})})$.

Also analogously to Def. 4, by $\to_{\mathrm{reg}^{+}}$ we denote the $\varrho^{\mathsf{S}}_{pos}$-eager rewrite strategy for Reg^{+}_{pos}.

$\to_{\mathrm{reg}^{+}}$-rewrite sequences on terms in $Ter(\boldsymbol{(\lambda^{\infty})})$ are related to $\to_{\mathrm{reg}^{+}}$-rewrite sequences on position-annotated terms via lifting (adding annotations) and projecting (dropping annotations). The proposition and the lemma below describe the connection between position-annotated $\to_{\mathrm{reg}^{+}}$-rewrite sequences and the concepts of binding, capturing, and binding–capturing chains. Then a lemma and the main theorem of this section are given.

▶ **Proposition 22.** For all $T \in Ter(\boldsymbol{\lambda^{\infty}})$ and positions $p, q \in Pos(T)$ it holds:

$$p \multimap q \iff ()^{\epsilon}_{\langle\rangle} T \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda x_1 \ldots x_n)^{q}_{p_1,\ldots,p_n} x_n \ \wedge\ p = p_n$$
$$p \leftarrow\!\!\cdot\ q \iff ()^{\epsilon}_{\langle\rangle} T \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda x_1 \ldots x_i \ldots x_n)^{q'}_{p_1,\ldots,p_i,\ldots,p_n} U \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda x_1 \ldots x_i)^{q}_{p_1,\ldots,p_i} x_i$$
$$\text{for some } i < n \text{ such that } p \in \{p_{i+1},\ldots,p_n\}$$

▶ **Lemma 23** (binding–capturing chains)**.** *For all $T \in Ter^{\infty}(\boldsymbol{\lambda})$ it holds:*

(i) *If $()^{\epsilon} T \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda x_1 \ldots x_n)^{q}_{p_1,\ldots,p_n} U$, then $p_1 \multimap q_2 \cdot\!\!\to p_2 \multimap \ldots \multimap q_n \cdot\!\!\to p_n$ holds for some $q_2,\ldots,q_n \in Pos(T)$.*

(ii) *If $p_1 \multimap q_2 \cdot\!\!\to p_2 \multimap \ldots \multimap q_n \cdot\!\!\to p_n$ is a binding–capturing chain in T, then there exist $r_1,\ldots,r_m,s \in Pos(T)$ with $m \geq n$ such that $()^{\epsilon} T \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda x_1 \ldots x_m)^{s}_{r_1,\ldots,r_m} U$ and furthermore $p_1,\ldots,p_n \in \{r_1,\ldots,r_m\}$ such that $p_1 < p_2 < \ldots < p_n = r_m$.*

▶ **Lemma 24** (infinite binding–capturing chains)**.** *Let T be an infinite λ-term, and let τ be an infinite $\to_{\mathrm{reg}^{+}}$-rewrite sequence $()T = (\lambda\vec{x}_0)T_0 \to_{\mathrm{reg}^{+}} (\lambda\vec{x}_1)T_1 \to_{\mathrm{reg}^{+}} \ldots$ with the property $\lim_{i\to\infty} |\vec{x}_i| = \infty$. Then there exists an infinite binding–capturing chain in T.*

Proof (Sketch). The assumed infinite $\to_{\mathrm{reg}^{+}}$-rewrite sequence can be lifted to one with position annotations $()^{\epsilon} T = (\lambda\vec{x}_0)^{\epsilon}_{\vec{p}_0} T_0 \to_{\mathrm{reg}^{+}} (\lambda\vec{x}_1)^{q_1}_{\vec{p}_1} T_1 \to_{\mathrm{reg}^{+}} (\lambda\vec{x}_2)^{q_2}_{\vec{p}_2} T_2 \to_{\mathrm{reg}^{+}} \ldots$ where q_i are positions and $\vec{p}_i = \langle p_1,\ldots,p_{m_i}\rangle$ vectors of positions. Due to $\lim_{i\to\infty} |\vec{x}_i| = \liminf_{i\to\infty} |\vec{x}_i| = \infty$ the sequence is of the form: $()^{\epsilon} T = (\lambda\vec{x}_{i_0})^{q_{i_0}}_{\vec{p}_{i_0}} T_{i_0} \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda\vec{x}_{i_1})^{q_{i_1}}_{\vec{p}_{i_1}} T_{i_1} \twoheadrightarrow_{\mathrm{reg}^{+}} (\lambda\vec{x}_{i_2})^{q_{i_2}}_{\vec{p}_{i_2}} T_{i_2} \twoheadrightarrow_{\mathrm{reg}^{+}} \ldots$ with $0 = |\vec{x}_{i_0}| < |\vec{x}_{i_1}| < |\vec{x}_{i_2}| < \ldots$ and such that $|\vec{x}_{i_j}| \leq |\vec{x}_k|$ holds for all $j,k \in \mathbb{N}$ with $k \geq i_j$. Since steps in $Reg^{+}_{\boldsymbol{pos}}$ remove position annotations only when the corresponding abstraction variable is dropped from the prefix in an $\to_{\mathsf{S}}$-step, it follows that $\vec{p}_{i_0} < \vec{p}_{i_1} < \vec{p}_{i_2} < \ldots$ holds

with respect to the prefix order $<$. Hence in the limit these vectors tend to an infinite sequence of positions $r = \langle r_1, r_2, r_3, \cdots \rangle$. Then Lemma 23, (23), can be used to show that the positions on r are the binder positions of an infinite binding–capturing chain in T. ◀

▶ **Theorem 25.** *An infinite λ-term is strongly regular if and only if it is regular and contains only finite binding–capturing chains.*

Proof (Sketch). Suppose that T is strongly regular. By Prop. 11, T is regular. Also, $ST^+(T)$ is finite. Let n be the length of the longest abstraction prefix in $ST^+(T)$. Then Lemma 23, (23), implies that the length of any binding–capturing chain in T is bounded by $n-1$.

Suppose that T is regular, but not strongly regular. Then $ST(T)$ is finite, while $ST^+(T)$ is infinite. Since the rewrite strategy $\to_{\mathrm{reg}^+}$ has branching degree ≤ 2 (branching only happens at sources of $\to_{@_i}$-steps), Kőnig's Lemma implies that there is an infinite $\to_{\mathrm{reg}^+}$-rewrite sequence $\tau : ()T = (\lambda\vec{x}_0)T_0 \to_{\mathrm{reg}^+} (\lambda\vec{x}_1)T_1 \to_{\mathrm{reg}^+} \dots$ that passes through distinct terms. By Lemma 10, (10), τ projects to a $\to_{\mathrm{reg}}$-rewrite sequence $\check{\tau} : ()T = (\lambda\vec{x}'_0)T_0 \twoheadrightarrow_{\mathrm{reg}} (\lambda\vec{x}'_1)T_1 \twoheadrightarrow_{\mathrm{reg}} \dots$ under $\twoheadrightarrow_{\mathrm{del}}$-rewrite sequences $(\lambda\vec{x}_i)T_i \twoheadrightarrow_{\mathrm{del}} (\lambda\vec{x}'_i)T_i$, for all $i \in \mathbb{N}$, that respectively shortening the length of the abstraction prefix. As $ST(T)$ is finite, $\check{\tau}$ passes only through finitely many terms. This contrast with τ can be used to show that prefix lengths of the terms on τ must tend to infinity. Due to this, Lemma 24 is applicable to τ, and yields the existence of an infinite binding–capturing chain in T. ◀

5 Expressibility by terms of the λ-calculus with μ

Having adapted (in Section 2) the concept of regularity for infinite λ-terms in two ways, we now obtain an expressibility result for one of these adaptations that is analogous to that in [4] for regular first-order trees with respect to rational expressions (or equivalently, μ-terms). We show that an infinite λ-term is strongly regular if and only if it is $\boldsymbol{\lambda}_\mu$-expressible.

We first define terms of $\boldsymbol{\lambda}_\mu$, the unfolding rewrite relation, and $\boldsymbol{\lambda}_\mu$-expressibility.

▶ **Definition 26** (CRS-representation for $\boldsymbol{\lambda}_\mu$)**.** The CRS-signature $\Sigma_{\lambda_\mu} = \Sigma_\lambda \cup \{\mathsf{mu}\}$ for $\boldsymbol{\lambda}_\mu$ extends Σ_λ by a unary function symbol mu. By $Ter(\boldsymbol{\lambda}_\mu)$ we denote the set of closed finite CRS-terms over Σ_{λ_μ} with the restriction that CRS-abstraction occurs only as an argument of the symbols abs or mu. By $Ter(\boldsymbol{(\lambda_\mu)})$ we denote the analogously defined set of terms over the signature $\Sigma_{(\lambda)} \cup \{\mathsf{mu}\}$. We consider the *$\mu$-unfolding rule* in informal and formal notation:

$$(\varrho^\mu): \ \mu x.M(x) \to M(\mu x.M(x)) \qquad \varrho^\mu: \ \mathsf{mu}([x]Z(x)) \to Z(\mathsf{mu}([x]Z(x)))$$

This rule induces the *unfolding rewrite relation* $\to_\mu$ on $Ter(\boldsymbol{\lambda}_\mu)$ and $Ter(\boldsymbol{(\lambda_\mu)})$. We say that a λ_μ-term M *expresses* an infinite λ-term V if $M \twoheadrightarrow_\mu V$ holds, that is, M unfolds to V via a typically infinite, strongly convergent $\to_\mu$-rewrite sequence (similar for terms in $Ter(\boldsymbol{(\lambda_\mu)})$). And an infinite λ-term T is *λ_μ-expressible* if there is a λ_μ-term M that expresses T.

We sketch some intuition for the proof, which proceeds by a sequence of proof-theoretic transformations. We focus on the more difficult direction. Let T be a strongly regular infinite λ-term. We want to extract a λ_μ-term M that expresses T from the finite $\to_{\mathrm{reg}^+}$-reduction graph G of T. We first obtain a closed derivation $\mathcal{D}$ of $()T$ in $\mathbf{Reg}_\mathbf{0}^+$. The derivation $\mathcal{D}$ can be viewed as a finite term graph that has G as its homomorphic image, and that does not exhibit horizontal sharing ([3, Sec. 4.3]). Such term graphs correspond directly to λ_μ-terms (analogous to [3]). In order to extract the λ_μ-term M corresponding to $\mathcal{D}$ from this derivation, we annotate it inductively to a λ_μ-term-annotated derivation $\hat{\mathcal{D}}$ with conclusion $()\, M : T$ in a proof system **Expr** that is a variant of $\mathbf{Reg}_\mathbf{0}^+$. Then it remains to show that M indeed

$$\frac{}{(\lambda\vec{x}y)\,y \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x}y)\,y}\;0 \qquad \frac{(\lambda\vec{x})\,M_0 \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,T_0 \qquad (\lambda\vec{x})\,M_1 \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,T_1}{(\lambda\vec{x})\,M_0\,M_1 \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,T_0\,T_1}\;@$$

$$\frac{(\lambda\vec{x}y)\,M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x}y)\,T}{(\lambda\vec{x})\,\lambda y.M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,\lambda y.T}\;\lambda \qquad \frac{(\lambda\vec{x})\,M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,T}{(\lambda\vec{x}y)\,M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x}y)\,T}\;\mathsf{S}\ \begin{array}{l}(\text{if the binding } \lambda y\\ \text{is vacuous}\\ \text{in } M \text{ and } T)\end{array}$$

$$\frac{(\lambda\vec{x})\,M(\mu f.M(f)) \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,T}{(\lambda\vec{x})\,\mu f.M(f) \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\,T}\;\mu$$

Figure 5 Proof system $\mathbf{Unf}^{\infty}$ for completely unfolding of λ_μ-terms into infinite λ-terms.

unfolds to T. For this we prove that $\hat{\mathcal{D}}$ unfolds to/gives rise to infinite derivations the variant systems $\mathbf{Expr}^{\infty}$ and $\mathbf{Unf}^{\infty}$, which witnesses infinite outermost rewrite sequences $M \twoheadrightarrow_\mu T$.

The CRS consisting of the rule ϱ^μ is orthogonal and fully-extended [13]. As a consequence of the result in [9] that outermost-fair strategies in orthogonal, fully extended iCRSs are normalizing, we obtain the following proposition.

▶ **Proposition 27.** Let $M \in Ter(\boldsymbol{\lambda}_\mu)$ and $T \in Ter(\boldsymbol{\lambda}^{\infty})$. If M expresses T, then there is an outermost $\to_\mu$-rewrite sequence of length $\leq\omega$ that witnesses $M \twoheadrightarrow_\mu T$, and T is the unique λ-term expressed by M. Analogously for prefixed terms in $\boldsymbol{\lambda}_\mu$ that express prefixed λ-terms. Hence the infinite outermost unfolding rewrite relation $\overset{\text{out}!}{\twoheadrightarrow}_\mu$ to infinite normal form defines a partial mapping from $Ter(\boldsymbol{\lambda}_\mu)$ to $Ter(\boldsymbol{\lambda}^{\infty})$, and from $Ter(\boldsymbol{(\lambda_\mu)})$ to $Ter(\boldsymbol{(\lambda^{\infty})})$.

The relation $\overset{\text{out}!}{\twoheadrightarrow}_\mu$ can be defined via derivability in the proof system $\mathbf{Unf}^{\infty}$ in Fig. 5: the existence of a possibly infinite derivation that is closed in the sense of Def. 12, and *admissible*, i.e. it is (S)-eager, and does not contain infinitely many consecutive instances of the rule (μ).

▶ **Proposition 28.** $\mathbf{Unf}^{\infty}$ is sound and complete w.r.t. $\overset{\text{out}!}{\twoheadrightarrow}_\mu$: For all $(\lambda\vec{x})T \in Ter(\boldsymbol{(\lambda^{\infty})})$ and $(\lambda\vec{x})\,M \in Ter(\boldsymbol{(\lambda_\mu)})$, $\vdash_{\mathbf{Unf}^{\infty}} (\lambda\vec{x})\,M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})T$ holds if and only if $(\lambda\vec{x})\,M \overset{\text{out}!}{\twoheadrightarrow}_\mu (\lambda\vec{x})T$.

▶ **Definition 29** (proof systems $\mathbf{Expr}$, $\mathbf{Expr}^{\infty}$, and $\mathbf{Expr}_\mu$, $\mathbf{Expr}_\mu^{\infty}$)**.** The natural-deduction-style proof system $\mathbf{Expr}$ has as its formulas abstraction prefixed λ_μ-terms annotated by infinite λ-terms, and the rules in Fig. 6. The system $\mathbf{Expr}_\mu$ has abstraction prefixed λ_μ-terms as formulas, and its rules arise from $\mathbf{Expr}_\mu$ by dropping the λ-terms. Derivability in these systems means the existence of a closed (no open assumptions), (S)-eager, finite derivation.

The variant $\mathbf{Expr}^{\infty}$ of $\mathbf{Expr}$ arises by replacing the rule (FIX) with the rule (μ) in Fig. 7. $\mathbf{Expr}_\mu^{\infty}$ arises from $\mathbf{Expr}_\mu$ analogously. A derivation in either of these systems is *admissible* if it does not contain infinitely many consecutive instances of (μ), and if it is (S)-eager in the sense of Def. 13. Derivability in these systems means the existence of an admissible derivation that is closed in the sense of Def. 12.

▶ **Lemma 30.** (i) *For every λ_μ-term M: $\vdash_{\mathbf{Expr}_\mu} ()M$ if and only if $\vdash_{\mathbf{Expr}_\mu^{\infty}} ()M$.*
(ii) *Every closed derivation in $\mathbf{Expr}_\mu^{\infty}$ contains only finitely many λ_μ-terms.*
(iii) *For every λ_μ-term M it holds: $\vdash_{\mathbf{Expr}_\mu} ()M$ if and only if there is no $\to_{\text{reg}^+}$-generated subterm of M (in $ST^+(M)$) of the form $()\mu x_0 \ldots x_n.x_0$ for $n \in \mathbb{N}$.*

Proof. For (30), in order to show "⇒" let $\mathcal{D}$ be a finite, closed, (S)-eager derivation in $\mathbf{Expr}_\mu$ with conclusion $()M$. By 'unfolding' this derivation through a process in which in each step:

$$\frac{}{(\lambda\vec{x}y)\ y : y}\ 0 \qquad \frac{(\lambda\vec{x}y)\ M : T}{(\lambda\vec{x})\ \lambda y.M : \lambda y.T}\ \lambda \qquad \frac{(\lambda\vec{x})\ M_0 : T_0 \qquad (\lambda\vec{x})\ M_1 : T_1}{(\lambda\vec{x})\ M_0\, M_1 : T_0\, T_1}\ @$$

$$\frac{(\lambda\vec{x})\ M : T}{(\lambda\vec{x}y)\ M : T}\ \mathsf{S}\ \text{(if the binding } \lambda y \text{ is vacuous)}$$

$$\frac{\begin{array}{c}[(\lambda\vec{x})\ \mathsf{c}_l : T]^l \\ \mathcal{D}_0 \\ (\lambda\vec{x})\ M(\mathsf{c}_l) : T\end{array}}{(\lambda\vec{x})\ \mu f.M(f) : T}\ \mathrm{FIX}, l \quad \begin{array}{l}(\text{if } |\mathcal{D}_0| \geq 1, \text{ and } |\vec{y}| \geq |\vec{x}| \text{ for all } (\lambda\vec{y})\ N : U \text{ on threads} \\ \text{from open assumptions } ((\lambda\vec{x})\ \mathsf{c}_l : T)^l \text{ down})\end{array}$$

Figure 6 Natural-deduction style proof system **Expr** for expressibility of infinite λ-terms by λ_μ-terms. The proof system $\mathbf{Expr}_{\boldsymbol{\mu}}$ for λ_μ-terms that express infinite λ-terms arises by dropping the colons ':' and the subsequent infinite λ-terms. Derivations in **Expr** and $\mathbf{Expr}_{\boldsymbol{\mu}}$ must be (S)-eager.

$$\frac{(\lambda\vec{x})\ M(\mu f.M(f)) : T}{(\lambda\vec{x})\ \mu f.M(f) : T}\ \mu$$

Figure 7 The proof system $\mathbf{Expr}^{\infty}$ for expressibility of λ-terms by λ_μ-terms arises from **Expr** by replacing the rule FIX with the rule μ. The proof system $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ for λ_μ-terms that express λ-terms arises from $\mathbf{Expr}^{\infty}$ by dropping the colons ':' and the subsequent infinite λ-terms. Admissible derivations in $\mathbf{Expr}^{\infty}$ and in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ do not have infinitely many consecutive instances of μ.

a subderivation of a bottommost instance of FIX

$$\frac{\begin{array}{c}[(\lambda\vec{y})\,\mathsf{c}_l]^l \\ \mathcal{D}_0(\mathsf{c}_l) \\ (\lambda\vec{y})\,N(\mathsf{c}_l)\end{array}}{(\lambda\vec{y})\,\mu f.N(f)}\ \mathrm{FIX}, l$$

is 'unfolded' into a subderivation

$$\frac{\begin{array}{c}[(\lambda\vec{y})\,\mathsf{c}_l]^l \\ \mathcal{D}_0(\mathsf{c}_l) \\ (\lambda\vec{y})\,N(\mathsf{c}_l)\end{array}}{[(\lambda\vec{y})\,\mu f.N(f)]}\ \mathrm{FIX}, l \qquad \frac{\begin{array}{c}\mathcal{D}_0(\mu f.N(f)) \\ (\lambda\vec{y})\,N(\mu f.N(f))\end{array}}{(\lambda\vec{y})\,\mu f.N(f)}\ \mu$$

in the limit a closed derivation $\mathcal{T}$ in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ is obtained with the same conclusion as $\mathcal{D}$. Furthermore, $\mathcal{T}$ does not contain infinitely many consecutive instances of μ, since the side-condition on (FIX) guarantees a guardedness condition analogous to Prop. 16. Hence $\mathcal{T}$ is a closed admissible derivation in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ with conclusion $()M$. For showing "$\Leftarrow$", suppose that $\mathcal{T}$ is a closed admissible derivation in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ with conclusion $()M$. Then there is a finite closed derivation $\mathcal{D}$ with the same conclusion in the variant system $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu},-}$ that does not require the side-condition part $|\mathcal{D}_0| \geq 1$ for instances of FIX. Via the process described above, $\mathcal{D}$ unfolds to a closed, (S)-eager derivation in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$, which has to be equal to $\mathcal{T}$, since closed (S)-eager derivations in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ are unique (due to the rules of this system). If $\mathcal{D}$ would not satisfy the guardedness condition described in Prop. 16, and therefore would also violate the mentioned side-condition part, for any of its FIX-instances, then $\mathcal{T}$ would not be admissible. It follows that $\mathcal{D}$ is a closed derivation in $\mathbf{Expr}_{\boldsymbol{\mu}}$ with conclusion $()M$.

For (30) note that by the argument for "$\Leftarrow$" in (30), every closed derivation in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu}}$ is the unfolding of a closed derivation in $\mathbf{Expr}_{\boldsymbol{\mu}}$, and that the unfolding process can produce only finitely many λ_μ-terms. Statement (30) follows by an easy analysis of closed derivations in $\mathbf{Expr}^{\infty}_{\boldsymbol{\mu},-}$ that violate the guardedness condition in Prop. 16 on any of its FIX-instances. ◀

▶ **Lemma 31.** *For all infinite prefixed λ-terms T, $\vdash_{\mathbf{Reg}^{+}_{0}} (\lambda\vec{x})T$ holds if and only if there exists a λ_μ-term M such that $\vdash_{\mathbf{Expr}} (\lambda\vec{x})\ M : T$.*

Proof (Sketch). Every derivation $\mathcal{D}$ in $\mathbf{Reg}^{+}_{0}$ with conclusion $(\lambda\vec{x})T$ can be annotated with appropriate λ_μ-terms, by induction on the derivation depth, thereby introducing appropriate

constants and exploiting the form of the rules in **Expr**, to a derivation $\hat{\mathcal{D}}$ in **Expr** with conclusion $(\lambda\vec{x})\ M : T$ and corresponding assumptions, for some prefixed λ_μ-term $(\lambda\vec{x})\, M$.

Conversely, the result of dropping the λ_μ-terms and the subsequent colons in a derivation $\mathcal{D}'$ in **Expr** results in a derivation $\check{\mathcal{D}}'$ in $\mathbf{Reg}_\mathbf{0}^+$. ◀

▶ **Example 32.** The derivation $\mathcal{D}_l$ in $\mathbf{Reg}_\mathbf{0}^+$ from Example 17, (i), on the left can be annotated, as described by Lemma 31 to obtain the following derivation $\hat{\mathcal{D}}_l$ in **Expr**:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(()\,\mathsf{c}_l : T)^l}{(\lambda x)\ \mathsf{c}_l : T}\,\mathsf{S}}{(\lambda xy)\ \mathsf{c}_l : T}\,\mathsf{S} \quad \dfrac{}{(\lambda xy)\ y : y}\,\mathsf{0}}{(\lambda xy)\ \mathsf{c}_l\, y : T\, y}\,@ \quad \dfrac{\dfrac{}{(\lambda x)\ x : x}\,\mathsf{0}}{(\lambda xy)\ x : x}\,\mathsf{S}}{(\lambda xy)\ \mathsf{c}_l\, y\, x : T\, y\, x}\,@}{(\lambda x)\ \lambda y.\mathsf{c}_l\, y\, x : T\, y\, x}\,\lambda}{()\ \lambda xy.\mathsf{c}_l\, y\, x : \lambda xy.T\, y\, x}\,\lambda}{()\ \mu f.\lambda xy.f\, y\, x : T}\,\mathrm{FIX}, u$$

Note that the λ_μ-term in the conclusion unfolds to T, the infinite λ-term in Fig. 1.

▶ **Theorem 33.** *The proof system* **Expr** *is sound and complete with respect to* $\twoheadrightarrow_\mu^!$*: for all expressions* $(\lambda\vec{x})\ M : T$ *of* λ_μ*-term-annotated, prefixed infinite* λ*-terms it holds that* $\vdash_{\mathbf{Expr}} (\lambda\vec{x})\ M : T$ *if and only if* $(\lambda\vec{x})\, M \twoheadrightarrow_\mu^! (\lambda\vec{x})\, T$*.*

Proof. For "⇒" let $\mathcal{D}$ be a closed, (S)-eager derivation in **Expr** with conclusion $(\lambda\vec{x})\ M : T$. By an unfolding process and arguments analogous as described in the proof of Lemma 30, $\mathcal{D}$ unfolds to a closed admissible prooftree $\mathcal{T}$ in $\mathbf{Expr}^\infty$ with the same conclusion. By changing all symbols ":" in $\mathcal{T}$ to "$\overset{\text{unf}}{\Longrightarrow}$", and distributing the abstraction prefixes in the expressions of $\mathcal{T}$ over "$\overset{\text{unf}}{\Longrightarrow}$", a closed admissible prooftree $\mathcal{T}'$ in $\mathbf{Unf}^\infty$ is obtained that has the conclusion $(\lambda\vec{x})\, M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\, T$. Then by Prop. 28 it follows that $(\lambda\vec{x})\, M \twoheadrightarrow_\mu^! (\lambda\vec{x})\, T$.

For "⇐", suppose that $(\lambda\vec{x})\, M \twoheadrightarrow_\mu^! (\lambda\vec{x})\, T$ holds. Then Prop. 28 entails that there is a closed admissible derivation $\mathcal{T}$ in $\mathbf{Unf}^\infty$ with the conclusion $(\lambda\vec{x})\, M \overset{\text{unf}}{\Longrightarrow} (\lambda\vec{x})\, T$. Since subderivations of closed admissible derivations in $\mathbf{Unf}^\infty$ are again such derivations, it follows by Prop. 28 and Prop. 27 that $\mathcal{T}$ does not contain more infinite λ-terms than λ_μ-terms. By dropping the symbols $\overset{\text{unf}}{\Longrightarrow}$ and the infinite λ-terms on the right, a closed admissible derivation $\mathcal{T}_\mu$ in $\mathbf{Expr}_\boldsymbol{\mu}^\infty$ is obtained. Due to Lemma 30, (30), $\mathcal{T}_\mu$, and hence $\mathcal{T}$, contains only finitely many λ_μ-terms. As $\mathcal{T}$ does not contain more infinite λ-terms than λ_μ-terms, it follows that $\mathcal{T}$ contains only finitely many formulas. By changing all symbols "$\overset{\text{unf}}{\Longrightarrow}$" in $\mathcal{T}$ into ":", and gathering the abstraction prefixes in the expressions of $\mathcal{T}$, a closed admissible prooftree $\mathcal{T}'$ in $\mathbf{Expr}^\infty$ with the conclusion $(\lambda\vec{x})\ M : T$ and only finitely many formulas are obtained.

Finally, similar as in the proof of Theorem 18, $\mathcal{T}'$ can be 'folded' into a finite closed derivation $\mathcal{D}'$ in **Expr** with conclusion $(\lambda\vec{x})\ M : T$ by introducing FIX-instances to cut off the derivation above the upper occurrence of a repetition (the side-condition on such instances of FIX is guaranteed due to admissibility of $\mathcal{T}'$). ◀

▶ **Theorem 34.** *An infinite* λ*-term is* λ_μ*-expressible if and only if it is strongly regular.*

Proof. For all infinite λ-terms T it holds:

$$\begin{aligned} T \text{ is } \lambda_\mu\text{-expressible} &\iff \exists M \in Ter(\boldsymbol{\lambda}_\mu).\ M \twoheadrightarrow_\mu^! T && \text{(by Prop. 27)} \\ &\iff \exists M \in Ter(\boldsymbol{\lambda}_\mu).\ \vdash_{\mathbf{Expr}} ()\ M : T && \text{(by Theorem 33)} \end{aligned}$$

$$\iff \vdash_{\mathbf{Reg}_0^+} ()T \qquad \text{(by Theorem 31)}$$
$$\iff T \text{ is strongly regular} \qquad \text{(by Theorem 18, (18))},$$

which establishes the statement of the theorem. ◀

As a consequence of Theorems 34 and 25 we obtain a theorem with our main results.

▶ **Theorem 35.** *For all infinite λ-terms the following statements are equivalent:*

(i) *T is λ_μ-expressible.*
(ii) *T is strongly regular.*
(iii) *T is regular, and it only contains finite binding–capturing chains.*

6 Generalization to $\boldsymbol{\lambda}_{\mathsf{letrec}}$ and practical perspectives

In [6] we undertook an in-depth study of expressibility in $\boldsymbol{\lambda}_{\mathsf{letrec}}$, and obtained the more general, but analogous result for full $\boldsymbol{\lambda}_{\mathsf{letrec}}$ instead of only for $\boldsymbol{\lambda}_\mu$. While there are significantly more technicalities involved, the structure of the proofs is analogous to here. Instead of demanding eager application of the scope-delimiting rules ϱ^{del} and ϱ^{S}, respectively, there we study λ-term decomposition $\to^{\mathbb{S}}_{\mathrm{reg}}$ and $\to^{\mathbb{S}}_{\mathrm{reg}^+}$ for arbitrary scope-delimiting strategies $\mathbb{S}$.

Concepts introduced here and in [6] have the potential to be practically relevant for the implementation of functional programming languages. In [8] we study various higher-order and first-order term-graph representations of cyclic λ-terms. Their definitions draw heavily on the decomposition rewrite systems in this paper. That is, every term in $\boldsymbol{\lambda}_{\mathsf{letrec}}$ can be translated into a finite first-order 'λ-term-graph' by applying the rewrite strategy $\to_{\mathrm{reg}^+}$ to the expressed strongly regular, infinite λ-term. Thereby vertices with the labels λ, @, S are created according to the kind of $\to_{\mathrm{reg}^+}$-step observed (plus variable occurrence vertices with label $\mathsf{0}$). The degree of sharing exhibited by λ-term-graphs can be analyzed with functional bisimulation. In [8] we identify a class of first-order representations with eager application of scope closure that faithfully preserves and reflects the sharing order on higher-order term graphs. This leads to an algorithm for efficiently determining the maximally shared form of a term in $\boldsymbol{\lambda}_{\mathsf{letrec}}$, which can be put to use in a compiler as part of an optimizing transformation.

Associated with this article is the report [7], which contains more details on Section 4. Also closely related is the report [6] about the more general case of expressibility in $\boldsymbol{\lambda}_{\mathsf{letrec}}$.

References

1 Zena M. Ariola and Stefan Blom. Cyclic Lambda Calculi. In Martin Abadi and Takayasu Ito, editors, *Proceedings of TACS'97*, volume 1281 of *LNCS*, pages 77–106. Springer, 1997.
2 Zena M. Ariola and Jan Willem Klop. Lambda Calculus with Explicit Recursion. *Information and Computation*, 139(2):154–233, 1997.
3 Stefan Blom. *Term Graph Rewriting – Syntax and Semantics*. PhD thesis, Vrije Universiteit Amsterdam, 2001.
4 Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25(2):95–169, 1983.
5 Jörg Endrullis, Clemens Grabmayer, Jan Willem Klop, and Vincent van Oostrom. On Equal μ-Terms. In I. Bethke, A. Ponse, and P.H. Rodenburg, editors, *Festschrift in Honour of Jan Bergstra*, Special Issue of TCS, 412 (28), pages 3175–3202. Elsevier, June 2011.
6 Clemens Grabmayer and Jan Rochel. Expressibility in the Lambda-Calculus with Letrec. Technical Report arXiv:1208.2383, `arxiv.org`, August 2012.
7 Clemens Grabmayer and Jan Rochel. Expressibility in the Lambda Calculus with μ. Technical Report arxiv:1304.6284, `arxiv.org`, 2013. Extends this article.

8 Clemens Grabmayer and Jan Rochel. Term Graph Representations for Cyclic Lambda Terms. In *Proc. of TERMGRAPH 2013*, number 110 in EPTCS, 2013. arXiv:1302.6338.

9 Jeroen Ketema and Jakob Grue Simonsen. Infinitary Combinatory Reduction Systems: Normalising Reduction Strategies. *Logical Methods in Computer Science*, 6(1:7):1–35, 2010.

10 Jeroen Ketema and Jakob Grue Simonsen. Infinitary Combinatory Reduction Systems. *Information and Computation*, 209(6):893 – 926, 2011.

11 Paul-André Melliès. *Description Abstraite des Systèmes de Réécriture (Thèse de doctorat)*. PhD thesis, l'Université Paris 7, December 1996.

12 Vincent van Oostrom. FD à la Melliès, February 1997. Vrije Universiteit Amsterdam.

13 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

A Homotopical Completion Procedure with Applications to Coherence of Monoids

Yves Guiraud[1], Philippe Malbos[2], and Samuel Mimram[3]

1 **Laboratoire Preuves, Programmes et Systèmes, INRIA, Université Paris 7**
yves.guiraud@pps.univ-paris-diderot.fr
2 **Institut Camille Jordan, Université Claude Bernard Lyon 1**
malbos@math.univ-lyon1.fr
3 **CEA, LIST**
samuel.mimram@cea.fr

Abstract

One of the most used algorithm in rewriting theory is the Knuth-Bendix completion procedure which starts from a terminating rewriting system and iteratively adds rules to it, trying to produce an equivalent convergent rewriting system. It is in particular used to study presentations of monoids, since normal forms of the rewriting system provide canonical representatives of words modulo the congruence generated by the rules. Here, we are interested in extending this procedure in order to retrieve information about the low-dimensional homotopy properties of a monoid. We therefore consider the notion of *coherent presentation*, which is a generalization of rewriting systems that keeps track of the cells generated by confluence diagrams. We extend the Knuth-Bendix completion procedure to this setting, resulting in a *homotopical completion procedure*. It is based on a generalization of *Tietze transformations*, which are operations that can be iteratively applied to relate any two presentations of the same monoid. We also explain how these transformations can be used to remove useless generators, rules, or confluence diagrams in a coherent presentation, thus leading to a *homotopical reduction procedure*. Finally, we apply these techniques to the study of some examples coming from representation theory, to compute minimal coherent presentations for them: braid, plactic and Chinese monoids.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases higher-dimensional rewriting, presentation of monoid, Knuth-Bendix completion, Tietze transformation, low-dimensional homotopy for monoids, coherence

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.223

1 Introduction

A monoid can be presented as the free monoid Σ_1^* on a set Σ_1 of *generators* quotiented by a congruence generated by a set of *relations* $\Sigma_2 \subseteq \Sigma_1^* \times \Sigma_1^*$. This data, called a *presentation* of the monoid, can be quite useful since it can provide a small description of it, from which various invariants can be computed, such as the homology of the monoid. For instance, the commutative monoid $\mathbb{N} \times \mathbb{N}$ admits the presentation $\langle \Sigma_1 \mid \Sigma_2 \rangle = \langle a, b \mid ba = ab \rangle$ with two generators and one relation. A way to show this result is to orient the relation as $ba \Rightarrow ab$ in order to obtain a *string rewriting system*. This rewriting system is easily checked to be terminating and confluent (there is no critical pair), and the normal forms are canonical representative of words modulo the congruence generated by the relation: here, normal forms are words of the form $a^m b^n$, which are in bijection with elements of the monoid $\mathbb{N} \times \mathbb{N}$.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 223–238

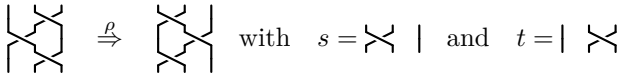
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


The Knuth-Bendix completion. This recipe for constructing presentations does not always work as easily. In particular, the rewriting system obtained by orienting arbitrarily the relations has no reason to be convergent (i.e. both terminating and confluent). However, it was observed by Knuth and Bendix [16] that by adding rules to the rewriting system, one can sometimes complete it into a finite convergent one. The procedure that they have formulated in order to perform this completion in good cases (the procedure is not guaranteed to terminate) is one of the most used tool in rewriting theory.

Tietze transformations. The starting point of the present work is the following observation: the Knuth-Bendix procedure operates by iteratively adding new relations, and this operation is a particular case of *Tietze transformation* [28]. These are basic operations that one can perform on a presentation, in such a way that they do not change the presented monoid, and one can always transform a presentation of a given monoid into another one by applying a series of such transformations; two presentations of the same monoid are thus called *Tietze-equivalent*. These transformations are of four kinds: add or remove a definable generator, and add or remove a derivable relation.

Adding generators. The Knuth-Bendix procedure only exploits one kind of transformations in order to complete a rewriting system: given a critical pair $v \overset{f}{\Leftarrow} u \overset{g}{\Rightarrow} w$ it adds a rule $h : v \Rightarrow w$ (or its converse), which is derivable since $h = f^{-1} \circ g$; in particular, adding it does not change the monoid presented by the rewriting system. Could the procedure be improved by also adding new generators during completion? On the theoretical level, an affirmative answer has been brought by Kapur and Narendran [14] who considered the usual Artin presentation Σ of the monoid $\mathbf{B}_3^+$ of positive braids with 3 strands (with its alternative graphical representation on the right):

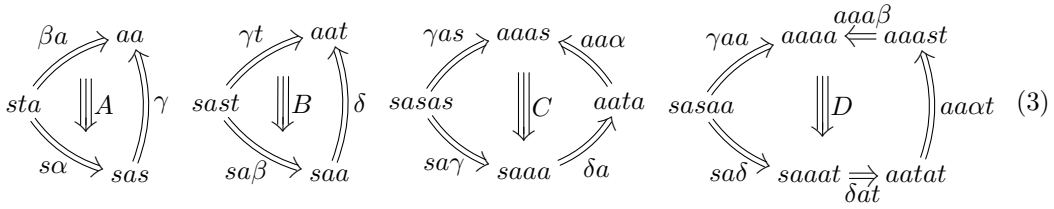
$$tst \overset{\rho}{\Rightarrow} sts \qquad \overset{\rho}{\Rightarrow} \quad \text{with} \quad s = \quad \text{and} \quad t = \tag{1}$$



They show that there exists no finite convergent string rewriting system, *with the same generators* s and t, that presents the monoid $\mathbf{B}_3^+$. However, they consider the string rewriting system Υ with three generators s, t and a new generator a (standing for the product st) and two relations $st \Rightarrow a$ and $sts \Rightarrow tst$. This rewriting system Υ is Tietze-equivalent to the rewriting system Σ, but applying the Knuth-Bendix completion procedure on it terminates, giving rise to the convergent rewriting system Υ' with s, t, a as generators, and rules

$$ta \overset{\alpha}{\Rightarrow} as, \qquad st \overset{\beta}{\Rightarrow} a, \qquad sas \overset{\gamma}{\Rightarrow} aa, \qquad saa \overset{\delta}{\Rightarrow} aat. \tag{2}$$

Thus, adding a superfluous generator has made completion possible. The reason why the completed rewriting system Υ' is Tietze-equivalent to the original rewriting system Σ can be understood by considering its four confluent critical branchings:



(3)

The cell $A : (\beta a) \Rrightarrow (s\alpha) \circ \gamma$ witnesses the fact that rule γ is superfluous since $\gamma = (s\alpha)^{-1} \circ (\beta a)$ and, similarly, the cell B proves that $\delta = (sa\beta)^{-1} \circ (\gamma t)$ is superfluous. Finally, the rule β

witnesses the fact that the generator a is superfluous (it is equivalent to st). We are left with the rule α where a has been substituted by st, i.e. $\rho : tst \Rightarrow sts$ in (1). As we will see, this example is far from being isolated, thus justifying the use of Tietze transformations as a central concept to study existing extensions and refinements of completion procedures, such as Pedersen's morphocompletion [23], or to introduce new ones.

A homotopical completion procedure. The four diagrams in (3) are the generators of an equivalence relation between rewriting paths: two paths with the same source and the same target are equal up to those diagrams. The previous discussion shows the importance of keeping track of those higher-dimensional cells, which carry information about the rewriting system (for example, if one wants to compute invariants of monoids such as homology groups). This motivates the use of a generalization of the notion of presentation, called *coherent presentation*, which takes in account the higher-dimensional information contained in *homotopy generators* (the diagrams in (3)), and of a generalization of the notion of Tietze transformation to this setting. The *homotopical completion procedure* extends the Knuth-Bendix procedure into a tool for computing coherent presentations, by keeping track of homotopy generators created when adding new rules.

A homotopical reduction procedure. The additional information contained in coherent presentations can also help one to reduce a presentation by removing superfluous generators, rules and homotopy generators. For instance, we have already mentioned that the cell A in (3) indicates that the rule γ is superfluous. Similarly, the rule β indicates that the generator a is superfluous since it is equivalent to the product st, and we will see that superfluous homotopy generators in (3) can be also removed by computing critical triples of the rewriting system. All these operations of removing superfluous data from a presentation are again examples of Tietze transformations. Based on these, we introduce here a *homotopical reduction procedure* for coherent presentations which minimizes a coherent presentation, such as one obtained from our homotopical completion procedure. The general idea of this work is thus to give ways to mutate presentations using Tietze transformations in order to come up with presentations satisfying various properties: convergence, coherence, minimality, etc.

Coherent presentations to compute invariants of monoids. Minimal presentations obtained in this way exhibit invariants of the monoid, in the sense that even though constructed from a particular presentation of the monoid, their number of generators, rules and homotopy generators do not depend on the presentation, only on the monoid. In particular, when the monoid is presented by a finite convergent presentation, the corresponding coherent presentation always has a finite number of homotopy generators. This important result of Squier's theory [26, 25, 27], further studied in subsequent works [19, 18, 17, 12], has enabled him to construct a finitely presented decidable monoid with no finite convergent presentation: as a consequence, rewriting is not universal to decide the word problem in decidable monoids.

Applications in algebra and representation theory. Coherent presentations also appear as a fundamental structure in representation theory (in particular through the examples of Artin and plactic monoids). One of the motivations of the results presented here is to apply constructive rewriting methods to compute coherent presentations for these algebraic structures arising in geometry. For instance, Tits' theorem [29] states that an Artin group has a coherent presentation where coherence cells are given by its parabolic subgroups of

rank 3 (a similar result holds for Artin monoids). The original proof relies on geometry, we give here a constructive methodology that has since been used to obtain a coherent presentation for any Artin monoid and group [9]. We also apply our completion methods to the plactic monoid, used in the representation theory of semisimple Lie algebras [21].

Contents of the paper. We introduce the notion of coherent presentation in Section 2, for which we formulate a homotopical completion-reduction procedure in Section 3, applied to various examples in Section 4. We should mention here that this works is part of a much larger general program aiming at studying the higher-dimensional properties of rewriting theory, see [10, 11, 22, 13, 12, 9] for example, based on Burroni's notion of polygraph [2].

2 Coherent presentations

The purpose of this section is to recall some classical material about rewriting systems and introduce some notions and notations from higher-dimensional rewriting. More details can be found in the mentioned references. In the following, we will assimilate the notion of string rewriting system to a presentation.

▶ **Definition 1.** A *presentation* $\Sigma = (\Sigma_1, \Sigma_2)$ consists of a set Σ_1 of *generators* and a set $\Sigma_2 \subseteq \Sigma_1^* \times \Sigma_1^*$ of *rewriting rules*. It is *finite* if both sets Σ_1 and Σ_2 are finite.

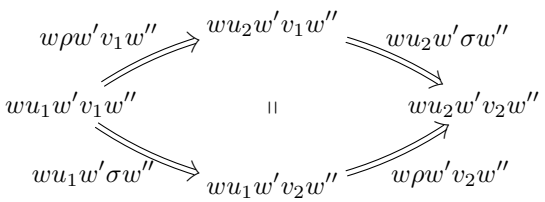
In the definition, Σ_1^* denotes the free monoid over Σ_1 (*words* over the alphabet Σ_1). We write $\rho : u \Rightarrow v$ for a rule $\rho = (u, v)$ in Σ_2. A word u *rewrites* to a word v, denoted by $u \Rightarrow_\Sigma v$, when there exist words w_1 and w_2 and a rule $\rho : u' \Rightarrow v'$ such that $u = w_1 u' w_2$ and $v = w_1 v' w_2$; the corresponding *rewriting step* is then denoted by $w_1 \rho w_2 : u \Rightarrow v$.

The *reduction graph* $\mathcal{G}_\Sigma$ of a presentation Σ is the graph whose vertices are words and whose edges are rewriting steps. We write $u \Rightarrow_\Sigma^* v$ when there exists a directed path from u to v in $\mathcal{G}_\Sigma$. A string rewriting system Σ is a *presentation* of a monoid $\mathbf{M}$ when $\mathbf{M}$ is isomorphic to the free monoid over Σ_1 quotiented by the congruence $\Leftrightarrow_\Sigma^*$ generated by the relations in Σ_2, i.e. $u \Leftrightarrow_\Sigma^* v$ whenever there exists a non-directed path from u to v in $\mathcal{G}_\Sigma$. We write $\langle a_1, \dots, a_n \mid u_1 \Rightarrow v_1, \dots, u_n \Rightarrow v_n \rangle$ for the monoid presented by the rewriting system Σ with $\Sigma_1 = \{a_1, \dots, a_n\}$ and $\Sigma_2 = \{\rho_1 : u_1 \Rightarrow v_1, \dots, \rho_n : u_n \Rightarrow v_n\}$.

2.1 Coherent presentations of monoids

We extend the notion of presentation in order to incorporate an equivalence between paths which is described by a set of homotopy generators, in the same way that rewriting rules specify an equivalence between words. In order to do so, we first have to explicitly consider the rewriting paths (and not only the convertibility relation, i.e. whether there exists a path between two words), which leads us to define the category generated by a rewriting system.

▶ **Definition 2.** Let Σ be a presentation. The *category* Σ_2^* *generated by* Σ has the words in Σ_1^* as objects, and the directed paths in $\mathcal{G}_\Sigma$ as morphisms, quotiented by the smallest congruence forgetting the order of rewriting steps at two disjoint positions, i.e. formally making the following diagrams commutative:

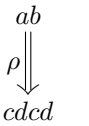


$$\text{for all } \begin{cases} u_1 \overset{\rho}{\Rightarrow} u_2 \text{ and } v_1 \overset{\sigma}{\Rightarrow} v_2 \text{ in } \Sigma_2 \\ w,\ w' \text{ and } w'' \text{ in } \Sigma_1^*. \end{cases}$$

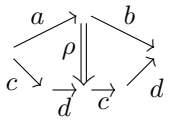
Equivalence of rewriting at disjoint positions is justified by local confluence (orthogonal branchings are never obstructions to confluence) and corresponds on the categorical side to the exchange axiom of 2-categories. The *groupoid* $\Sigma_2^\top$ *generated by the presentation* Σ is the category defined similarly, with inverses added for each morphism: morphisms are non-directed paths in $\mathcal{G}_\Sigma$ and we write $f^{-1} : v \Rightarrow u$ for a path $f : u \Rightarrow v$ taken backwards.

▶ **Example 3.** Consider the presentation Σ with $\Sigma_1 = \{s, t, a\}$ as generators, and whose rules in Σ_2 are the four rules of (2). The following composite of rewriting steps is a morphism in Σ_2^*: $(sa\gamma) \circ (\delta a) \circ (aa\alpha) : sasas \Rightarrow aaas$; it occurs in the border of the cell C in (3). Similarly, the composite $(s\alpha) \circ \gamma \circ (\beta a)^{-1} : sta \Rightarrow sta$ is a morphism in the groupoid $\Sigma_2^\top$, which is the border of the cell A in (3).

Given a morphism $f : u \Rightarrow v$ and words w_1, w_2 in Σ_1^*, we write $w_1 f w_2 : w_1 u w_2 \Rightarrow w_1 v w_2$ for the morphism f "extended by context". This enables us to equip the category Σ_2^* with a structure of monoidal category: given two morphisms $f : u \Rightarrow v$ and $f' : u' \Rightarrow v'$, we can define a morphism $f \otimes f' : (u \otimes u') \Rightarrow (v \otimes v')$ which corresponds to performing the two rewriting paths f and f' in parallel. Formally, the tensor product is defined on objects by concatenation $(u \otimes v = uv)$ and on morphisms $f : u \Rightarrow v$ and $f' : u' \Rightarrow v'$ by $f \otimes f' = (fu') \circ (vf')$. This monoidal structure on the category Σ_2^* induces a shift in the dimension of objects. However, this category is a monoidal category, which equivalently amounts to say that it is a 2-category with only one 0-cell, the objects of Σ_2^* being the 1-cells and the morphisms of Σ_2^* being the 2-cells. From a diagrammatic point of view, this means that



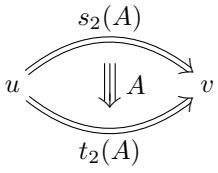
should actually be drawn as



In the following, we adopt this convention for the dimension of cells, but we keep drawing diagrams as the one on the left, since those are closer to diagrams traditionally used in rewriting theory. This also applies to $\Sigma_2^\top$, which is also be considered as a 2-category (with invertible 2-cells) in the following.

We can now introduce the notion of coherent presentation, by enriching presentations with a suitable specified set of confluence 3-cells. Below, two 2-cells are said to be *parallel* when they have the same source and the same target 1-cells.

▶ **Definition 4.** A *(finite) extended presentation* $(\Sigma, s_2, t_2, \Sigma_3)$ consists of a (finite) presentation Σ, together with a (finite) set Σ_3 of 3-*cells* (or *homotopy generators*) and two maps $s_2, t_2 : \Sigma_3 \to \Sigma_2^\top$ associating, to each 3-cell A, parallel 2-cells of $\Sigma_2^\top$ which are respectively its *source* $s_2(A) : u \Rightarrow v$ and its *target* $t_2(A) : u \Rightarrow v$ (cf. the diagram on the right).



A *homotopy relation* on $\Sigma_2^\top$ is an equivalence relation $\equiv$ on parallel 2-cells which is stable under context and composition:
– for any f and g in $\Sigma_2^\top$ and any u and v in Σ_1^*, $f \equiv g$ implies $ufv \equiv ugv$,
– for any $h : u' \Rightarrow u$, $f, g : u \Rightarrow v$ and $k : v \Rightarrow v'$ in $\Sigma_2^\top$, $f \equiv g$ implies $h \circ f \circ k \equiv h \circ g \circ k$.
In particular, given an extended presentation Σ, we write $\equiv_{\Sigma_3}$ for the smallest homotopy relation containing Σ_3.

▶ **Definition 5.** A *(finite) coherent presentation* is a (finite) extended presentation Σ such that the homotopy relation generated by Σ_3 is the homotopy relation on $\Sigma_2^\top$ containing every pair of parallel 2-cells.

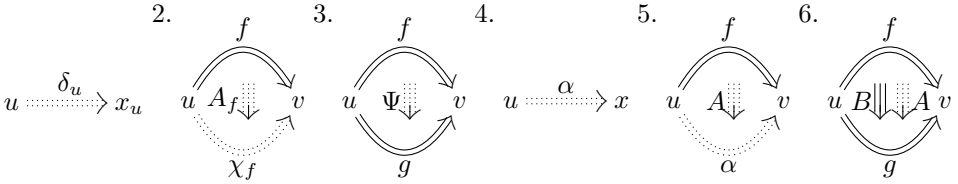
▶ **Example 6.** The presentation Σ of Example 3 can be extended into a coherent presentation with the three diagrams A, B and C of (3) as set of 3-cells. For instance, we have $s_2(A) = \beta a$ and $t_2(A) = s\alpha \circ \gamma$. Notice that, since the 2-cells of $\Sigma_2^\top$ are invertible, different choices for the source and target of 3-cells could still give a coherent presentation, such as $s_1(A) = (\beta a)^{-1} \circ (s\alpha)$ and $t_1(A) = \gamma^{-1}$.

In the same way that rewriting systems present monoids, a coherent presentation presents a 2-category. Namely, given a coherent presentation Σ, one can define a 2-category, denoted by $\Sigma_2^\top/\Sigma_3$, as the 2-category $\Sigma_2^\top$ whose 2-cells have been quotiented by the homotopy relation $\equiv_{\Sigma_3}$. Notice that this 2-category always has its 2-cells invertible.

2.2 Transformations of coherent presentations

Starting from a non-convergent presentation of a monoid, the Knuth-Bendix procedure provides a (convergent) presentation on the same set of generators, but a monoid can also admit other presentations with different sets of generators. The notion of Tietze transformation [28] describes elementary transformations (adding and removing definable generators or derivable rules) on presentations, leaving unchanged the presented monoid. Moreover, they are complete in the sense that two presentations of the same monoid are related by Tietze transformations. In [9], a corresponding notion has been introduced for extended presentations, defined as the composites of the following elementary transformations:

1. **add a generator** $\mathcal{T}_u^+$: for u in Σ_1^*, add x_u to Σ_1 and $\delta_u : u \Rightarrow x_u$ to Σ_2,
2. **add a relation** $\mathcal{T}_f^+$: for $f : u \Rightarrow v$ in $\Sigma_2^\top$, add $\chi_f : u \Rightarrow v$ to Σ_2 and $A_f : f \Rrightarrow \chi_f$ to Σ_3,
3. **add a 3-cell** $\mathcal{T}_{(f,g)}^+$: for $f \equiv_{\Sigma_3} g$ in $\Sigma_2^\top$, add $\Psi : f \Rrightarrow g$ to Σ_3,
4. **remove a generator** $\mathcal{T}_x^-$: for $\alpha : u \Rightarrow x$ in Σ_2, with $x \in \Sigma_1$ and $u \in (\Sigma_1 \setminus \{x\})^*$, remove x and α and replace x by u in the relations and 3-cells and α by 1_u in the 3-cells,
5. **remove a relation** $\mathcal{T}_\alpha^-$: for $A : f \Rrightarrow \alpha$ in Σ_3, with $\alpha \in \Sigma_2$ and $f \in (\Sigma_2 \setminus \{\alpha\})^*$, remove α and A and replace α by f in the 3-cells,
6. **remove a 3-cell** $\mathcal{T}_A^-$: for $A : f \Rrightarrow g$ in Σ_3 with $f \equiv_{\Sigma_3 \setminus \{A\}} g$, remove A.



A (finite) *Tietze transformation* is a (finite) composite of elementary Tietze transformations. The notion of Tietze-equivalence on presentations can be generalized to extended presentations: Σ and Υ are *Tietze-equivalent* extended presentations if they are Tietze-equivalent as presentations and when there is an equivalence of categories $(\Sigma_2^\top/\Sigma_3) \cong (\Upsilon_2^\top/\Upsilon_3)$, see [9]. In particular, coherent presentations of a same monoid are Tietze-equivalent. As in the case of presentations, we have for extended presentations:

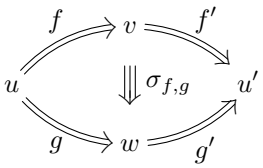
▶ **Theorem 7** ([9]). *Two (finite) extended presentations are Tietze-equivalent if, and only if, there exists a (finite) Tietze transformation between them.*

2.3 Computing coherent presentations

We investigate a method to construct of coherent presentations from convergent ones, based on Squier's theory. We suppose fixed a presentation Σ. A *branching* of Σ is a pair

$(f : u \Rightarrow v, g : u \Rightarrow w)$ of 2-cells of Σ_2^* with a common source. Such a branching is *local* when f and g are both rewriting steps, *Peiffer* when it is of the form $(f'v, ug')$ or $(vf', g'u)$ for some rewriting steps $f' : u \Rightarrow u', g' : v \Rightarrow v'$, and *overlapping* when it is not Peiffer and f and g are distinct. A branching is *critical* when it is overlapping and minimal for the order generated on branchings by $(f, g) \preccurlyeq (ufv, ugv)$, for any words u and v. A branching $(f, g) : u \Rightarrow (v, w)$ is *confluent* when there exist a pair of 2-cells $f' : v \Rightarrow u'$ and $g' : w \Rightarrow u'$ in Σ_2^*. We say that Σ is (*locally*) *confluent* when all of its (local) branchings are confluent. We say that Σ is *convergent* when it terminates and it is confluent. By Newman's lemma, local confluence is equivalent to confluence of critical branchings for terminating rewriting systems. This result can be reformulated in the setting of coherent presentations as follows.

A *family of generating confluences of* Σ is a set of 3-cells over $\Sigma_2^\top$ that contains, for every critical branching (f, g) of Σ, one 3-cell whose shape is as in the diagram on the right. If Σ is confluent, it always admits at least one family of generating confluences. Given a convergent presentation Σ, we denote by $\mathcal{S}(\Sigma)$ the extended presentation obtained from Σ by adjunction of a chosen family of generating confluences of Σ. The presentation $\mathcal{S}(\Sigma)$ is only defined up to that choice, but two families of generating confluences give Tietze-equivalent extended presentations [12]. Squier proved in [27] the following result.



▶ **Theorem 8** (Squier's theorem). *Let Σ be a (finite) convergent presentation of a monoid $\mathbf{M}$. The extended presentation $\mathcal{S}(\Sigma)$ is a (finite) coherent and convergent presentation of $\mathbf{M}$.*

Several examples of this construction are given in [18]. Squier proved that the property, for a finite presentation of a monoid $\mathbf{M}$, to be extendable into a finite coherent presentation is an invariant of $\mathbf{M}$, that is, one given finite presentation of $\mathbf{M}$ is extendable into a finite coherent presentation if, and only if, all of them are [27]. However, there are finitely presented decidable monoids with no finite coherent presentation (such an example was exhibited by Squier). For such a monoid, starting with a finite presentation, there is no hope to obtain a finite convergent presentation, by using the Knuth-Bendix procedure or other methods, with the same set of generators or another one. Conversely, if the Knuth-Bendix procedure terminates on a finite presentation, then it can be extended into a finite coherent presentation.

3 Homotopical completion and reduction procedures

As seen in Section 2.3, Squier's theorem extends a convergent presentation into a coherent one. With the Knuth-Bendix completion procedure, those are the two basic ingredients of the homotopical completion procedure we present, extended by a homotopical reduction procedure whose goal is to eliminate superfluous cells.

3.1 The homotopical completion procedure

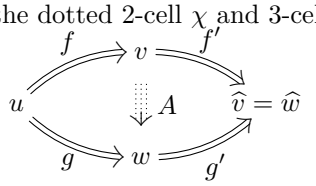
This procedure, denoted by $\mathcal{HC}$, interleaves the Knuth-Bendix completion and Squier's theorem to produce a coherent and convergent presentation from a terminating presentation: it examines the critical branchings one by one, potentially adding 2-cells to reach a convergent presentation, but also 3-cells that tend towards forming a coherent presentation.

Let Σ be a terminating presentation, seen as an extended presentation with no 3-cell. Thereafter, we always assume that termination is due to a fixed total termination order. For every critical branching $(f, g) : u \Rightarrow (v, w)$ of Σ, the procedure $\mathcal{HC}$ computes 2-cells

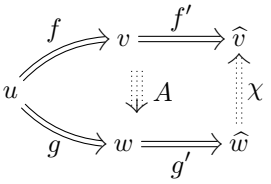
$f' : v \Rightarrow \widehat{v}$ and $g' : w \Rightarrow \widehat{w}$ in Σ_2^*, where $\widehat{v}$ and $\widehat{w}$ are some normal forms for v and w, respectively. There are two possibilities:

- if $\widehat{v} = \widehat{w}$, the dotted 3-cell A is added, as in situation (a),
- otherwise, for example if $\widehat{v} < \widehat{w}$, the Tietze transformation $\mathcal{T}^+_{g'^{-1} \circ g^{-1} \circ f \circ f'}$ is applied to add the dotted 2-cell χ and 3-cell A, as in (b).

(a)



(b)



The adjunction of new 2-cells can create new critical branchings: the $\mathcal{HC}$ procedure iterates this operation until it reaches, potentially after an infinite time, a stable extended presentation $\mathcal{HC}(\Sigma)$. From a computational point of view, an application of Squier's theorem to the result of the Knuth-Bendix completion on Σ would require to compute again all the critical branchings explored during completion, when the $\mathcal{HC}$ procedure computes 3-cells during completion. The properties of the Knuth-Bendix procedure and Squier's theorem induce the following result.

▶ **Theorem 9.** *Let Σ be a terminating presentation of a monoid $\mathbf{M}$. The extended presentation $\mathcal{HC}(\Sigma)$ is a coherent and convergent presentation of $\mathbf{M}$ and it is finite if, and only if, the presentation Σ is finite and the homotopical completion procedure terminates.*

▶ **Example 10.** The Kapur-Narendran presentation

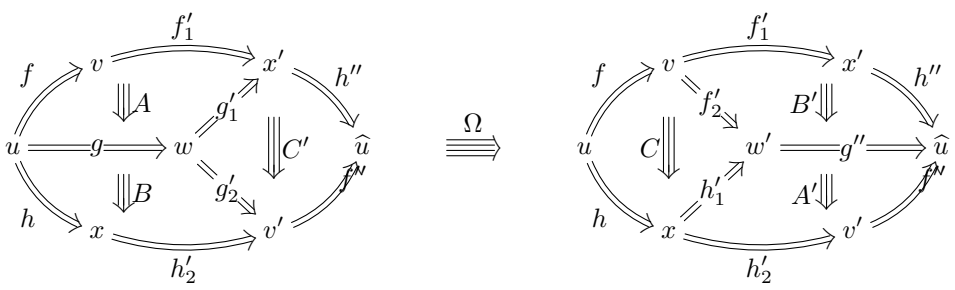
$$\mathbf{B}_3^+ = \langle s, t, a \mid \alpha : ta \Rightarrow as,\ \beta : st \Rightarrow a \rangle$$

has two non-confluent critical branchings, resulting in the adjunction of the 2-cells γ and δ as in (2) and the 3-cells A and B as in (3). The 2-cells γ and δ generate two new critical branchings that are confluent: the $\mathcal{HC}$ procedure adds two extra 3-cells C and D and terminates with this finite coherent and convergent presentation of the monoid $\mathbf{B}_3^+$.

3.2 An optimized homotopical completion procedure

The $\mathcal{HC}$ procedure computes a coherent and convergent presentation that contains, in general, superfluous 3-cells, in the sense that they are not necessary to relate all the parallel 2-cells. To eliminate them, we apply a *homotopical reduction* mechanism in dimension 3: it computes the critical triple branchings to produce relations between 3-cells and to eliminate some of them by Tietze transformations. A *critical triple branching* (f, g, h) is a triple of distinct rewriting steps with common source, such that each one overlaps with at least one of the other two, and that is minimal for the order $\preccurlyeq$ generated by relations $(f, g, h) \preccurlyeq (ufv, ugv, uhv)$ for every such triples (f, g, h) and words u, v.

Let Σ be a convergent and coherent presentation. The *homotopical reduction in dimension* 3 builds, for each critical triple branching (f, g, h) of Σ, a 4-cell Ω with shape

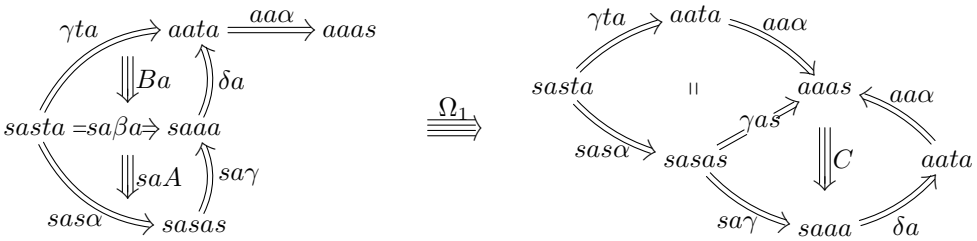


as follows. We consider the branching (f, g) and use confluence to get f_1' and g_1' and, then, coherence to build the 3-cell A. We proceed similarly with the branchings (g, h) and (f, h). Then, for the branching (f_1', f_2'), we use convergence to get g'' and h'' with $\widehat{u}$ as common target, and the 3-cell B' by coherence. We do the same operation with (h_1', h_2') to get A'. Finally, we get the 3-cell C' by coherence. The source and the target of Ω are made of generating 3-cells of Σ in context: they have shape uXv where X is a generating 3-cell and u and v are words. If one of those generating 3-cell appears only once and in an empty context ($u = v = 1$), then Ω is used as a definition of X in terms of the other 3-cells: X is removed by a Tietze transformation.

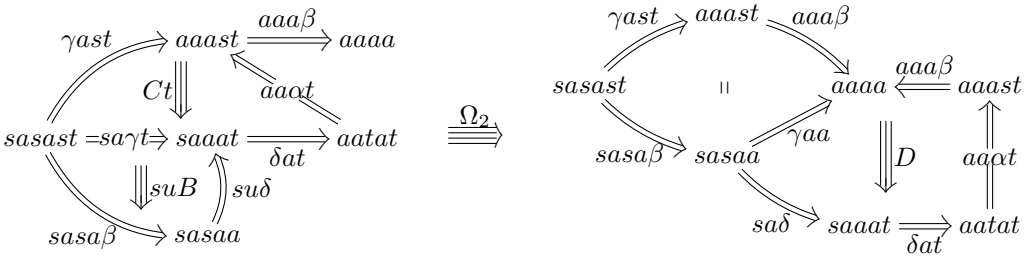
A coherent and convergent presentation on which no 3-cell can be removed by homotopical reduction in dimension 3 is called *reduced*. The *optimized homotopical completion procedure* $\overline{\mathcal{HC}}$ applies homotopical reduction in dimension 3 after $\mathcal{HC}$. Since the procedure acts by Tietze transformations only, we get:

▶ **Theorem 11.** *Let Σ be a terminating presentation of a monoid $\mathbf{M}$. The extended presentation $\overline{\mathcal{HC}}(\Sigma)$ is a reduced coherent and convergent presentation of $\mathbf{M}$, that is finite if, and only if, the presentation Σ is finite and the homotopical completion procedure $\mathcal{HC}$ terminates.*

▶ **Example 12.** After the $\mathcal{HC}$ procedure is applied to the Kapur-Narendran presentation of the monoid $\mathbf{B}_3^+$, we have four critical triple branchings, overlapping on the words *sasta*, *sasast*, *sasasas* and *sasasaa*. On *sasta*, we get the 4-cell



This 4-cell proves that C is superfluous in the coherent presentation: it appears only once in the boundary of Ω_1, in an empty context (unlike A and B). Then, we consider the critical triple branching with source *sasast*:



For the same reasons, the 4 cell Ω_2 removes D, leaving only the 3-cells A and B to form a reduced coherent and convergent presentation of the monoid $\mathbf{B}_3^+$. The other two critical triple branchings on words *sasasas* and *sasasaa* do not generate any other relation. Indeed, since the relations are weight-homogeneous (they relate words with the same weight, where s and t have weight 1 and a has weight 2), all the words that occur in the 4-cells corresponding to *sasasas* and *sasasaa* have weight 10 and 11, respectively. Since A and B have respective weights 4 and 5, their potential occurrences in those 4-cells must be in non-empty contexts.

3.3 The homotopical completion-reduction procedure

After the $\overline{\mathcal{HC}}$ procedure, we get a reduced coherent and convergent presentation of the considered monoid. Its underlying presentation is, in general, not minimal since homotopical completion has potentially adjoined superfluous 2-cells to get confluence. However, for each of those extra 2-cells, a 3-cell has also been added to fill the corresponding confluence diagram: a Tietze transformation can be used to remove both of them.

Given a coherent presentation Σ, we call *homotopical reduction in dimension* 2 the following process. For each 3-cell A of Σ_3, its source and target are made of reduction steps $u\alpha v$, where α is a generating 2-cell and u and v are words. If one such α appears only once and in an empty context ($u = v = 1$), both α and A are removed by a Tietze transformation. On the special case of $\overline{\mathcal{HC}}(\Sigma)$, every superfluous 2-cell appears once and in an empty context in the boundary of its associated 3-cell. The *homotopical completion-reduction procedure* $\mathcal{HCR}$ applies homotopical reduction in dimension 2 after $\overline{\mathcal{HC}}$. Since the procedure acts by Tietze transformations only, we get:

▶ **Theorem 13.** *Let Σ be a terminating presentation of a monoid $\mathbf{M}$. The extended presentation $\mathcal{HCR}(\Sigma)$ is a coherent presentation of $\mathbf{M}$, whose underlying presentation is contained in Σ, and it is finite if, and only if, the homotopical completion procedure terminates.*

▶ **Example 14.** After the $\overline{\mathcal{HC}}$ procedure is applied to the Kapur-Narendran presentation of the monoid $\mathbf{B}_3^+$, we have a coherent presentation with three generators s, t and a, four 2-cells α, β, γ and δ and two 3-cells A and B, corresponding to the adjunction of γ and δ respectively. They are removed by the $\mathcal{HCR}$ procedure, yielding a coherent presentation of $\mathbf{B}_3^+$ with the 2-cells α and β only, and no 3-cell. Informally, for two words on $\{s, t, a\}$ that represent the same element in the monoid $\mathbf{B}_3^+$, there is only one proof of their equality modulo α and β.

3.4 Completion and reduction on generators

As proved by Kapur and Narendran [14], the introduction of superfluous generators can be necessary for completion to terminate. These generators can of course be added by hand before the completion, but we briefly indicate here a possible heuristic, based on algebraic properties observed on the examples in Section 4. Indeed, in those cases, it always helps completion to add generators of the *quasicenter* of each submonoid. More precisely, for a given presentation Σ of a monoid $\mathbf{M}$, we seek minimal elements u of Σ_1^* such that $uX = Xu$ holds in $\mathbf{M}$ for a maximal subset X of Σ_1. Such a property is possible to observe during completion: one computes the products ux and xu for u a word of bounded length and x a generator. If $uX = Xu$ for a set X of generators, one adds a new generator (u) and a relation $u \Rightarrow (u)$. Moreover, the cardinal of X seems to determine a way to extend to (u) the termination order used for completion (see 4.3).

Whether the generators have been added before or during homotopical completion, one can remove them at the end. Indeed, each superfluous generator (u) comes with a defining relation $\alpha : u \Rightarrow (u)$, so that a Tietze transformation removes both of them (and replaces (u) by u and α by the identity in the boundary of the other 2-cells and 3-cells). Applied to the result of the $\mathcal{HCR}$ completion on the Kapur-Narendran presentation of the monoid $\mathbf{B}_3^+$, this contracts the obtained coherent presentation (with no 3-cell) to Artin presentation of the monoid $\mathbf{B}_3^+$, proving that this is also a coherent presentation with no 3-cell.

4 Applications

The results mentioned in this section were obtained with the help of a prototype implementation; an online version (unfortunately much slower than the offline one) is available[1].

4.1 The braid monoid

The Artin presentation. The *monoid* $\mathbf{B}_n^+$ *of positive braids on* n *strands* is defined by

$$\mathbf{B}_n^+ = \langle s_1, \dots, s_{n-1} \mid s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1} \text{ for } 1 \leq i < n-1,\ s_i s_j = s_j s_i \text{ for } |i-j| \geq 2 \rangle \quad (4)$$

This presentation, called *Artin presentation*, is known to be minimal, so that one wants to compute a minimal coherent presentation of the monoid $\mathbf{B}_n^+$ by extending it. In [29], Tits proved a result that implies that a coherent presentation is given by 3-cells whose boundaries are in one of the Artin submonoid of rank 3 of $\mathbf{B}_n^+$, i.e. the boundary of each 3-cell is made of copies of the three relations involving only three given distinct generators. As a direct consequence, Artin presentation with no 3-cell is a coherent presentation of the monoid $\mathbf{B}_3^+$, but this result fails to say anything about $\mathbf{B}_4^+$ and does not give an explicit description of the coherence cells of $\mathbf{B}_n^+$ for $n \geq 5$. Unfortunately, homotopical completion cannot be used either in practice because it does not terminate on Artin presentation: indeed, as proved by Kapur and Narendran, any orientation of a relation $sts = tst$ generates a relation $stsst^k = ts^{k+1}ts$ for every $k \geq 1$ that must be contained in every convergent presentation [14].

The Kapur-Narendran presentation. As far as we know, the adjunction of the superfluous generator for $\mathbf{B}_3^+$, as seen in the introduction, has not been studied for $\mathbf{B}_n^+$ with $n > 3$. There are several possible generalizations, but we define the *Kapur-Narendran presentation* of $\mathbf{B}_n^+$ as the one obtained from Artin presentation by adjunction of superfluous generators corresponding to a Coxeter element for each Artin submonoid of $\mathbf{B}_n^+$, namely all the products $s_{i_1} \cdots s_{i_k}$ for every $1 \leq i_1 < \cdots < i_k < n$. Our experiments lead to positive results for the cases $n = 4$ and $n = 5$, see Table 1. We have also tested the Kapur-Narendran presentation on well-known generalizations of the braids monoids known as Artin monoids and got a finite coherent and convergent presentation for the Artin monoids of types B_2, B_3, B_4 and F_4 (the braid monoid $\mathbf{B}_n^+$ is the Artin monoid of type A_{n-1}). An open question is to determine if the Kapur-Narendran presentation yields a finite coherent and convergent presentation for any braid monoid and, more generally, for other types of Artin monoids.

The Garside presentation. The Kapur-Narendran presentation is contained in a bigger presentation called the *Garside presentation* [8]. For $\mathbf{B}_3^+$, the Garside presentation is obtained from Artin one by adjunction of superfluous generators (st), (ts) and (sts) corresponding to products st, ts and sts respectively: the element sts is the generator of the quasicenter of $\mathbf{B}_3^+$ and the elements s, t, st and ts are all its divisors. On the Garside presentation, the homotopical completion procedure produces a finite coherent and convergent presentation with five generators, twelve relations and 24 3-cells; the corresponding normal forms are known as *Deligne's normal forms* [5]. Deligne has proved in [6] that this coherent presentation of $\mathbf{B}_3^+$ can be reduced to one with six relations

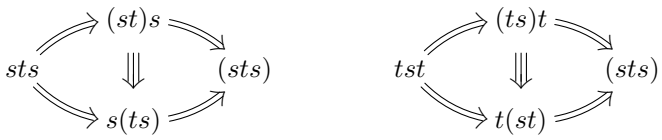
$$st \Rightarrow (st) \quad ts \Rightarrow (ts) \quad s(ts) \Rightarrow (sts) \quad t(st) \Rightarrow (sts) \quad (st)s \Rightarrow (sts) \quad (ts)t \Rightarrow (sts)$$

[1] http://www.pps.univ-paris-diderot.fr/~smimram/rewr/

Table 1 Results of experiments indicating, for various sets of generators, the number of generators, relations (before and after completion), and homotopy generators (before and after homotopy reduction by 4-cells) of the completed rewriting system. Values marked "†" arise from theoretical computations, and "?" indicate computations too big to be performed in reasonable time with our prototype implementation.

Coherent presentations						
Monoid	Presentation	Gen.	Rel.	Rel. comp.	Hom. gen.	Hom. gen. red.
$\mathbf{B}_3^+$	Artin	2	1	$\infty^\dagger$	$\infty^\dagger$	$0^\dagger$
	Kapur-Narendran	3	2	4	4	2
	Brieskorn-Saito	3	2	4	6	2
	Garside	5	4	12	24	8
$\mathbf{B}_4^+$	Artin	3	3	$\infty^\dagger$	$\infty^\dagger$	$1^\dagger$
	Kapur-Narendran	7	7	47	356	31
	Brieskorn-Saito	7	7	46	378	35
$\mathbf{B}_5^+$	Artin	4	6	$\infty^\dagger$	$\infty^\dagger$	$4^\dagger$
	Kapur-Narendran	15	17	692	48260	?
	Brieskorn-Saito	15	17	598	28384	?
$\mathbf{P}_2 = \mathbf{Ch}_2$	Knuth	2	2	2	1	1
	Column	3	3	3	1	1
$\mathbf{P}_3$	Knuth	3	8	11	27	23
	Column	7	12	22	42	30
$\mathbf{P}_4$	Knuth	4	20	$\infty^\dagger$	$\infty^\dagger$	$?^\dagger$
	Column	15	31	115	621	212
$\mathbf{P}_5$	Column	31	66	531	6893	?

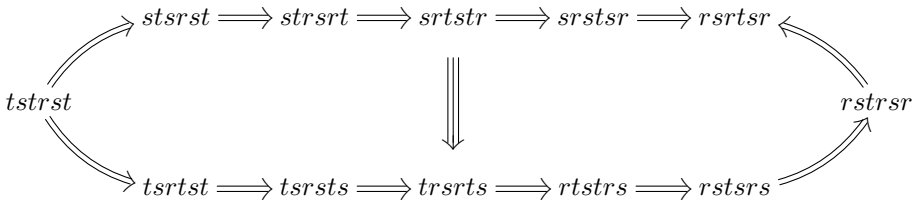
and two 3-cells:



The homotopical completion-reduction, applied to the Garside presentation, gives a new, constructive proof of this result [9]. In fact, it goes even further: the 3-cells are used to remove the relations $(st)s \Rightarrow (sts)$ and $(ts)t \Rightarrow (sts)$ and, then, the relations $st \Rightarrow (st)$, $ts \Rightarrow (ts)$ and $s(ts) \Rightarrow (sts)$ remove the generators (st), (ts) and (sts). This leaves the generators s and t, the relation $t(st) \Rightarrow (sts)$, projected onto $tst \Rightarrow sts$, and no coherence cell, yielding another proof that Artin presentation with no 3-cell is a coherent presentation of $\mathbf{B}_3^+$.

The Garside presentation exists for every monoid $\mathbf{B}_n^+$ and, more generally, for every Artin monoid: in the spherical case (such as $\mathbf{B}_n^+$), its generators are made of the generator of the quasicenter of every Artin submonoid, plus all of its divisors. On this presentation, the homotopical completion-reduction procedure also applies, extending Deligne's result to non-spherical Artin monoids. Moreover, in [9], Gaussent and the first two authors apply homotopical reduction further to get an explicit coherent presentation of every Artin monoid, thus, in particular, giving a constructive proof of Tits's result. For the particular case of $\mathbf{B}_4^+$, the homotopical completion-reduction procedure gives a (minimal) coherent presentation made of Artin presentation with exactly one coherence cell, known as the *Zamolodchikov*

relation:



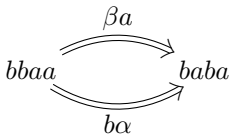
The Brieskorn-Saito presentation. For the monoid $\mathbf{B}_3^+$, it is defined by the adjunction to Artin presentation of a generator (sts) for sts [1]. This presentation is known in general for Artin monoids and obtained by adjunction of the generator (when it exists) of the quasicenter of each Artin submonoid. Those generators produce special normal forms that, up to our knowledge, are not yet linked to a convergent presentation. Contrarily to Garside's generators, Brieskorn-Saito's generators come in a finite number for every Artin monoid, motivating the research for a finite convergent presentation on those generators to give a solution to the still-open word problem for general Artin groups. Our experiments show that, on the Brieskorn-Saito presentation, the homotopical completion procedure gives a finite coherent and convergent presentation for the monoids $\mathbf{B}_3^+$, $\mathbf{B}_4^+$ and $\mathbf{B}_5^+$, but also for other Artin monoid such as the ones of type B_3 and, interestingly, of type $\tilde{A}_2$: this last example is an Artin monoid of *affine* type, for which the Garside presentation is infinite.

4.2 The plactic monoid

The Knuth presentation. The *plactic monoid* $\mathbf{P}_n$ *of rank* n is given by the Knuth presentation:

$$\mathbf{P}_n = \langle x_1, \dots, x_n \mid x_j x_i x_k = x_j x_k x_i \text{ for } i < j \leq k \text{ and } x_i x_k x_j = x_k x_i x_j \text{ for } i \leq j < k \rangle.$$

This monoid originates in the work of Schensted [24], Knuth [15] and Lascoux and Schützenberger [20]. It has found several applications, such as in representation theory [21] because of its strong connection to Young tableaux: semistandard Young tableaux correspond to elements of the plactic monoid and Schensted's insertion algorithm gives a way to compute normal forms for the Knuth presentation of the plactic monoid.



In the case $n = 2$, the Knuth presentation has two generators $x_1 = a$ and $x_2 = b$ and two relations $\alpha : baa \Rightarrow aba$ and $\beta : bba \Rightarrow bab$. This terminating presentation (for the deglex order generated by $a < b$ for example) is already convergent: the homotopical completion procedure yields a homotopy basis with exactly one coherence cell depicted on the right. Moreover, the convergent presentation has no critical triple branching: hence the computed coherent presentation of the monoid $\mathbf{P}_2$ is minimal.

In the case $n = 3$, with generators a, b and c, the Knuth presentation has eight relations, three pairs corresponding to the three plactic submonoids over two of the three generators, plus two relations involving all three generators: $\gamma : cab \Rightarrow acb$ and $\delta : bca \Rightarrow bac$. For the monoid $\mathbf{P}_3$, the Knuth presentation is not confluent anymore (on words $cbba$, $ccbaa$, $ccbab$) and homotopical completion adds three more relations: $\varepsilon : cbab \Rightarrow bcba$, $\varphi : cbaba \Rightarrow cacba$ and $\psi : cbcba \Rightarrow cbacb$. At the end, we get a finite coherent and convergent presentation with 27 3-cells, corresponding to all the critical branchings. The presentation also has 29 triple critical branchings, and homotopical reduction uses four of them to eliminate of four 3-cells. Then, the removal of the three extra relations and their corresponding coherence

cells, added by completion, yields a homotopy basis with 20 coherence cells for the Knuth presentation of the monoid $\mathbf{P}_3$.

For higher values of n, the homotopical completion procedure cannot succeed on the Knuth presentation. Indeed, as in the case of braid monoids, a proof similar to the one of Kapur and Narendran for the monoid $\mathbf{B}_3^+$ shows that the infinite family of relations $cbc^kdca = cbac^kdc$, for every natural number k, must be part of any convergent presentation of the monoid $\mathbf{P}_n$.

The column presentation. The analogy with Young tableaux leads to introduce a finite number of superfluous generators to the Knuth presentation of $\mathbf{P}_n$, representing all the possible columns in semistandard Young tableaux: one generator $(x_{i_k} \cdots x_{i_1})$ for every possible $1 < k \leq n$ and $1 \leq i_1 < \cdots < i_k \leq n$, together with the corresponding defining relation $x_{i_k} \cdots x_{i_1} \Rightarrow (x_{i_k} \cdots x_{i_1})$. The column generators have an important property in plactic monoids: indeed, the center (and the quasicenter) of the plactic monoid $\mathbf{P}_n$ is generated by exactly one element: $x_n \cdots x_1$. Thus the column generators for $\mathbf{P}_n$ are exactly the generators of the quasicenters of all the plactic submonoids of $\mathbf{P}_n$.

From the column presentation, homotopical completion yields a finite coherent and convergent presentation of $\mathbf{P}_n$ (as in [3]). In particular, for the monoid $\mathbf{P}_4$, we get the following construction. Starting with the Knuth presentation with four generators and 20 relations, we add the eleven column generators (ba, ca, cb, da, db, dc, cba, dba, dca, dcb, $dcba$) and the corresponding relations to get 15 generators and 31 relations. Homotopical completion results in a finite coherent and convergent presentation with 115 relations and 621 3-cells. Then, homotopical reduction in dimension 3 uses the triple critical branchings to reduce the number of 3-cells to 212. The removal of the 84 relations and 3-cells added during homotopical completion, then of the eleven superfluous generators and their defining relations, finally produces a coherent presentation made of the Knuth presentation of the monoid $\mathbf{P}_4$ and 128 3-cells.

4.3 The Chinese monoid

The standard presentation. The *Chinese monoid* $\mathbf{Ch}_n$ *of rank* n is defined by

$$\mathbf{Ch}_n = \langle x_1, \ldots, x_n \mid x_j x_k x_i = x_k x_i x_j = x_k x_j x_i \text{ for } i \leq j \leq k \rangle.$$

It is a variant of the plactic monoid discovered in [7]. For $n = 2$, the Chinese monoid coincides with the plactic monoid $\mathbf{P}_2$: its standard presentation (with the orientation $baa \Rightarrow aba$ and $bba \Rightarrow bab$) is convergent and, with the same 3-cell as $\mathbf{P}_2$, forms a coherent presentation of $\mathbf{Ch}_2$. For higher values of n, the presentation of $\mathbf{Ch}_n$ (with the orientation $x_k x_i x_j \Rightarrow x_j x_k x_i$ and $x_k x_j x_i \Rightarrow x_j x_k x_i$) is not convergent anymore, but it can be finitely completed (without change of generators) by adjunction of the relations $x_k x_j x_k x_i \Rightarrow x_k x_i x_k x_j$ for $1 \leq i < j < k \leq n$. The homotopical completion-reduction yields a coherent presentation made of the standard presentation extended with 12 3-cells for $\mathbf{Ch}_3$, 56 for $\mathbf{Ch}_4$ and 176 for $\mathbf{Ch}_5$.

The quasicentral presentation. The (quasi)center of the monoid $\mathbf{Ch}_n$ is generated by the element $x_n x_1$ [4]. Thus, the generators of the quasicenters of all the Chinese submonoids of $\mathbf{Ch}_n$ are exactly the elements $x_j x_i$ for $1 \leq i < j \leq n$. Our experiments, conducted up to $n = 5$, show that the adjunction of those elements as superfluous generators still allow completion to reach a finite convergent presentation. Moreover, the obtained finite

convergent presentation gives a rewriting-based procedure to compute the *column normal form* [4]. For the completion, a special order has to be chosen (corresponding to the column normal form), such as a weight lexicographic order, where each x_i has weight 1 and $(x_j x_i)$ has weight 2, and with an order on generators that satisfies $(x_l x_i) > (x_k x_j)$ if $i \leq j \leq k \leq l$ with $i \neq j$ or $k \neq l$. This last inequality can be determined automatically from the fact that $(x_k x_i)$ commutes with $l - i$ elements and $(x_k x_j)$ with $k - j$ and, by assumption, we have $l - i > k - j$.

5 Conclusion

We have generalized the Knuth-Bendix completion procedure to coherent presentations, which has enabled us to formulate a reduction procedure. Some practical outcomes have been investigated, providing constructive results about presentations of braid, plactic and Chinese monoids. These procedures have been implemented in a proof-of-concept software, and much work remain to be done in order to better understand the structures put to use and how to efficiently manipulate them. The idea of adding superfluous generators seems very promising, but we have only been able to provide heuristics to do so which have to be refined and supported by more experiments. Finally, the approach developed here handles generators, relations and homotopy generators uniformly; its likely extension to higher dimensions will be investigated in future works, in relation with methods for constructing minimal presentations of algebraic structures.

References

1. E. Brieskorn and K. Saito. Artin-Gruppen und Coxeter-Gruppen. *Invent. Math.*, 17:245–271, 1972.
2. A. Burroni. Higher-dimensional word problems with applications to equational logic. *Theoret. Comput. Sci.*, 115(1):43–62, 1993.
3. A. Cain, R. Gray, and A. Malheiro. Finite Gröbner–Shirshov bases for plactic algebras and biautomatic structures for plactic monoids. Preprint, 16 pages, 2012.
4. J. Cassaigne, M. Espie, F. Hivert, D. F. Krob, and J.-C. Novelli. The Chinese monoid. *Internat. J. Algebra Comput.*, 11(3):301–334, 2001.
5. P. Deligne. Les immeubles des groupes de tresses généralisés. *Invent. Math.*, 17:273–302, 1972.
6. P. Deligne. Action du groupe des tresses sur une catégorie. *Invent. Math.*, 128(1):159–175, 1997.
7. G. Duchamp and D. Krob. Plactic-growth-like monoids. In *Words, languages and combinatorics, II (Kyoto, 1992)*, pages 124–142. World Sci. Publ., River Edge, NJ, 1994.
8. F. Garside. The braid group and other groups. *Quart. J. Math. Oxford Ser.*, 20:235–254, 1969.
9. S. Gaussent, Y. Guiraud, and P. Malbos. Coherent presentations of Artin groups. Preprint, 68 pages, arXiv:1203.5358v2, hal-00682233, 2013.
10. Y. Guiraud and P. Malbos. Higher-dimensional categories with finite derivation type. *Theory Appl. Categ.*, 22:No. 18, 420–478, 2009.
11. Y. Guiraud and P. Malbos. Coherence in monoidal track categories. *Math. Structures Comput. Sci.*, 22(6):931–969, 2012.
12. Y. Guiraud and P. Malbos. Higher-dimensional normalisation strategies for acyclicity. *Adv. Math.*, 231(3-4):2294–2351, 2012.
13. Y. Guiraud and P. Malbos. Identities among relations for higher-dimensional rewriting systems. *Semin. Congr.*, 26:145–161, 2013.

14 D. Kapur and P. Narendran. A finite Thue system with decidable word problem and without equivalent finite canonical system. *Theoret. Comput. Sci.*, 35(2-3):337–344, 1985.

15 D. E. Knuth. Permutations, matrices, and generalized young tableaux. *Pacific J. Math.*, 34(3):709–727, 1970.

16 D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra (Proc. Conf., Oxford, 1967)*, pages 263–297. Pergamon, Oxford, 1970.

17 Y. Kobayashi. Complete rewriting systems and homology of monoid algebras. *J. Pure Appl. Algebra*, 65(3):263–275, 1990.

18 Y. Lafont. A new finiteness condition for monoids presented by complete rewriting systems (after Craig C. Squier). *J. Pure Appl. Algebra*, 98(3):229–244, 1995.

19 Y. Lafont and A. Prouté. Church-Rosser property and homology of monoids. *Math. Structures Comput. Sci.*, 1(3):297–326, 1991.

20 A. Lascoux and M.-P. Schützenberger. Le monoïde plaxique. In *Noncommutative structures in algebra and geometric combinatorics (Naples, 1978)*, volume 109 of *Quad. "Ricerca Sci."*, pages 129–156. CNR, Rome, 1981.

21 P. Littelmann. A plactic algebra for semisimple Lie algebras. *Adv. Math.*, 124(2):312–331, 1996.

22 S. Mimram. Computing Critical Pairs in 2-Dimensional Rewriting Systems. In C. Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 227–242, 2010.

23 J. Pedersen. Morphocompletion for one-relation monoids. In *RTA*, volume 355 of *LNCS*, pages 574–578. Springer, 1989.

24 C. Schensted. Longest increasing and decreasing subsequences. *Canad. J. Math.*, 13:179–191, 1961.

25 C. C. Squier. Word problems and a homological finiteness condition for monoids. *J. Pure Appl. Algebra*, 49(1-2):201–217, 1987.

26 C. C. Squier and F. Otto. The word problem for finitely presented monoids and finite canonical rewriting systems. In *Proc. of RTA'87*, volume 256 of *LNCS*, pages 74–82. Springer, Berlin, 1987.

27 C. C. Squier, F. Otto, and Y. Kobayashi. A finiteness condition for rewriting systems. *Theoret. Comput. Sci.*, 131(2):271–294, 1994.

28 H. Tietze. Über die topologischen Invarianten mehrdimensionaler Mannigfaltigkeiten. *Monatsh. Math. Phys.*, 19(1):1–118, 1908.

29 J. Tits. A local approach to buildings. In *The geometric vein*, pages 519–547. Springer, 1981.

Extending Abramsky's Lazy Lambda Calculus: (Non)-Conservativity of Embeddings

Manfred Schmidt-Schauß[1], Elena Machkasova[2], and David Sabel[1]

1 **Goethe-Universität, Frankfurt, Germany**
schauss,sabel@ki.informatik.uni-frankfurt.de
2 **Division of Science and Mathematics, University of Minnesota, Morris, U.S.A.**
elenam@morris.umn.edu

Abstract

Our motivation is the question whether the lazy lambda calculus, a pure lambda calculus with the leftmost outermost rewriting strategy, considered under observational semantics, or extensions thereof, are an adequate model for semantic equivalences in real-world purely functional programming languages, in particular for a pure core language of Haskell. We explore several extensions of the lazy lambda calculus: addition of a seq-operator, addition of data constructors and case-expressions, and their combination, focusing on conservativity of these extensions. In addition to untyped calculi, we study their monomorphically and polymorphically typed versions. For most of the extensions we obtain non-conservativity which we prove by providing counter-examples. However, we prove conservativity of the extension by data constructors and case in the monomorphically typed scenario.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases lazy lambda calculus, contextual semantics, conservativity

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.239

1 Introduction

We are interested in reasoning about the semantics of lazy functional programming languages such as Haskell [11], in particular in semantical equivalences of expressions and, as a more general issue, in correctness of program translations and transformations. As a notion of expression equivalence in a calculus, we employ *contextual equivalence* which identifies expressions iff they cannot be distinguished when observing convergence to WHNFs in any surrounding context. Contextual equivalence is coarser than the (syntactical) conversion equality, and provides a more useful language model due to its maximal set of equivalences.

However, complexity of a language makes analyses and reasoning hard, so it is advantageous to find conceptually simpler sublanguages which also permit reasoning about equivalences in the superlanguage. As a starting point we may use the pure core language, say L^{α}_{Hcore}, of Haskell [12], which is a Hindley-Milner polymorphically typed call-by-need lambda calculus extended by data constructors, case-expressions, `seq` for strict evaluation and `letrec` to model recursive bindings and sharing. The semantics of such extended lambda calculi have been analyzed in several papers [20, 9, 10, 19, 18].

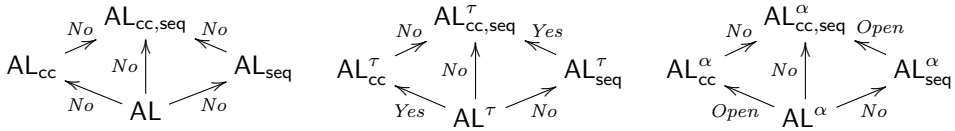
However, even this language has a rich syntax and thus one may ask whether there are simpler and/or smaller languages which can be used to reason about (parts of) Haskell. The issue of transferring the equivalence question is as follows: given two expressions s_1, s_2 in a calculus L, in which cases is it possible to decide the semantic equivalence $s_1 \sim s_2$

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 239–254

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


by transferring the equivalence question for s_1, s_2 into a smaller or conceptually simpler language L_{simple}, using the proof methods in L_{simple}? There are three (standard) types of transfer steps: (i) from a typed language L^τ into its untyped language L (which may be larger). Since we use contextual equivalence, in general $s_1 \sim_L s_2$ implies $s_1 \sim_{L^\tau} s_2$ for equally typeable expressions s_1, s_2, and thus this is a valid transfer, however, some equivalences may be lost. (ii) from a language L into a sublanguage L_{sub} by the removal of a syntactic construction possibility. Since now all expressions of L_{sub} are also L-expressions, the desired implication $s_1 \sim_L s_2 \implies s_1 \sim_{L^\tau} s_2$ exactly corresponds to conservativity of the inclusion w.r.t. equivalence. (iii) transferring the question to an isomorphic language L'.

We consider four calculi in this paper: Abramsky's lazy lambda calculus AL and its extensions $\mathsf{AL}_{\mathsf{seq}}, \mathsf{AL}_{\mathsf{cc}}, \mathsf{AL}_{\mathsf{cc,seq}}$ with `seq`, with `case` and constructors, and the combination of the two extensions, resp. We also consider variants of these calculi with monomorphic (τ-superscript) and polymorphic (α-superscript) types. We analyze whether natural embeddings between the calculi are conservative w.r.t. contextual equivalence in the calculi.
Our results can be depicted as follows, where *Yes*/*No* indicates a conservative (non-conservative, resp.) embedding, and *Open* indicates that the question is still unresolved.



A common pattern is that the removal of `seq` makes the embeddings non-conservative.
A powerful commonly used proof technique in all the calculi under consideration is based on Howe's method [6, 7], which shows that contextual equivalence coincides with *applicative bisimilarity* which equates expressions if they cannot be distinguished by first evaluating them, then applying their results to arguments, and then using this experiment co-inductively. Our improvement, which is valid since the languages are deterministic, is a so-called *AP_i-context lemma*, which means that expressions are equivalent iff their termination behavior is identical when applying them in all possible ways to finitely many arbitrary arguments.

Our results are of help for equivalence reasoning in L^α_{Hcore} considering implication chains for the justification of equivalences. The first one starts with transferring to the untyped core-language L_{Hcore}, then removing the syntactic construct `letrec` (and changing call-by-need to call-by-name), justified in [17, 18], arriving at $\mathsf{AL}_{\mathsf{cc,seq}}$. Then our results and counterexamples for the four untyped calculi come into play, where the conclusion is that further transfer steps appear impossible, in particular that AL [1] cannot be justified as equivalence checking calculus via this implication chain. The second implication chain takes another potential route: the first step is monomorphising the core language, then removing the `letrec`, adding `Fix`, and again changing the reduction strategy to call-by-name, arriving at the calculus $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$. We believe that both implications of equivalence are correct, but a formal proof is future work. Then, for the calculi $\mathsf{AL}^\tau_{\mathsf{cc}}$, $\mathsf{AL}^\tau_{\mathsf{seq}}$, AL^τ, we got negative as well as positive results. A further step could then be omitting the monomorphic types as well, which gives a valid implication chain from $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$ to $\mathsf{AL}^\tau_{\mathsf{seq}}$ and to $\mathsf{AL}_{\mathsf{seq}}$, but again there is no justification for AL and AL^τ as equivalence checking calculi for L^α_{Hcore}. Thus our results show that calculus for the transfer is $\mathsf{AL}_{\mathsf{cc,seq}}$, and under the correctness assumptions above, also $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$, $\mathsf{AL}^\tau_{\mathsf{seq}}$, and $\mathsf{AL}_{\mathsf{seq}}$. Focusing on the direct relation between the minimal calculi compared with L^α_{Hcore}, and taking into account our counterexamples in the paper, $\mathsf{AL}^\tau_{\mathsf{cc}}$ and $\mathsf{AL}_{\mathsf{cc}}$ are ruled out by examples s_7, s_8. However, it is still possible that AL or AL^τ can be used as equivalence checking calculi L^α_{Hcore} (although there are very few nontrivial equivalences there), which is strongly related to the open problem of whether there exist Böhm-like trees for AL (see Problem 18 in [22]).

Related Work. Our approach follows the general setup laid out e.g. in [4, 16] which consider the questions of relative expressivity between programming languages. In difference to [4], we use applicative bisimilarity and the AP_i-context lemma as a proof technique, and explore different calculi extensions. The closest work to ours is [13] that shows, in particular, that the extension of a monomorphically typed PCF with sum and product types and with Girard/Reynolds polymorphic types is conservative. They also show that extending PCF with a "convergence tester" by second-order polymorphic types is conservative. However, they do not (dis-)prove conservativity of adding the convergence tester to PCF and also do not consider an untyped case, or the pure lambda calculus.

Adding `seq` to call-by-need/call-by-name functional languages is investigated in several papers (e.g. [4, 5, 8]). For the lazy lambda calculus and its extension by `seq` an example in [4] can be adapted to show non-conservativity (see Theorem 4.3). It is well-known that in full Haskell `seq` makes a difference: The usual free theorems [23] break under the addition of `seq` [8], and the monad laws do not hold for the `IO`-monad if the first argument of `seq` is allowed to be of an `IO`-type [14]. [18, 19] provide a counterexample showing non-conservativity of adding `seq` to the lazy lambda calculus with data-constructors and case expressions.

Research on calculi extensions with case and constructors also including studies of untyped calculi is [2, 3, 21]. In [2] the addition of case and constructors to a basic calculus is explored. However, that calculus significantly differs from our ones in several points, e.g. it permits full η-reduction. [3] and [21] study an extension of a lambda calculus with surjective pairs. However, these works are incomparable to our approach since they use an axiomatic approach to equality instead of a rewriting and observational one.

Structure of the paper. In Sect. 2 we introduce a common notion for program calculi together with the notion of contextual equivalence. In Sect. 3 we briefly introduce the lazy lambda calculus AL and its three extensions $\mathsf{AL}_{\mathsf{cc}}, \mathsf{AL}_{\mathsf{seq}}, \mathsf{AL}_{\mathsf{cc,seq}}$. Conservativity of embeddings between the untyped calculi is refuted in Sect. 4. In Sect. 5 the monomorphically typed variants of the calculi are investigated. Sect. 6 presents the analysis of polymorphically typed calculi. We conclude in Sect. 7. Due to space reasons not all proofs are given, but they can be found in the technical report [15].

2 Preliminaries

We define our notion of a program calculus in an abstract way:

▶ **Definition 2.1.** A *typed deterministic program calculus* (TDPC) is a tuple $(\mathcal{E}, \mathcal{C}, \to_D, \mathcal{A}, \mathcal{T})$ where $\mathcal{E}$ is the (nonempty) set of *expressions*, such that every $s \in \mathcal{E}$ has a type $T \in \mathcal{T}$. We write $\mathcal{E}_T$ for the expressions of type T, and assume $\mathcal{E}_T \neq \emptyset$. We also assume that $\mathcal{E}$ can be divided into *closed* and *open* expressions, where $\mathcal{E}^c$ denotes the set of closed expressions. We use s, t, r, a, b, d to denote expressions and x, y, z, u to denote variables. $\mathcal{C}$ is the set of *contexts*, such that every $C \in \mathcal{C}$ is a function $C : \mathcal{E}_T \to \mathcal{E}_{T'}$ where $T, T' \in \mathcal{T}$. With $\mathcal{C}_{T,T'}$ we denote the contexts that are functions from $\mathcal{E}_T$ to $\mathcal{E}_{T'}$. We assume that $\mathcal{C}$ contains the identity function for every type $T \in \mathcal{T}$, and that $\mathcal{C}$ is closed under composition, i.e. iff $C_1 \in \mathcal{C}_{T_2,T_3}$ and $C_2 \in \mathcal{C}_{T_1,T_2}$ then also $(C_1 \circ C_2) \in \mathcal{C}_{T_1,T_3}$. We denote the application of contexts C to an expression $s \in \mathcal{E}$ by $C[s]$. The *standard reduction relation* $\to_D \subseteq (\mathcal{E} \times \mathcal{E})$ must be: (i) deterministic: $s_1 \to_D s_2$ and $s_1 \to_D s_3$ implies $s_2 = s_3$, where $=$ is syntactical equivalence (which usually also identifies α-equivalent expressions); (ii) type preserving: $s_1 \to_D s_2$ implies that s_1 and s_2 are of the same type; (iii) closedness-preserving: if s_1 is closed and $s_1 \to_D s_2$, then s_2 is closed. The set $\mathcal{A} \subseteq \mathcal{E}$ are the *answers* of the calculus, which are usually irreducible values or specific kinds of normal forms. We use v to range over answers.

An untyped calculus can also be presented as a typed one, by adding a single type called "expression". However, we simply write $(\mathcal{E}, \mathcal{C}, \to_D, \mathcal{A})$ for such a calculus.

We denote the transitive-reflexive closure of $\to_D$ by $\xrightarrow{*}_D$, and $\xrightarrow{n}_D$ with $n \in \mathbb{N}_0$ means n reductions. We define the notions of convergence, contextual approximation, and contextual equivalence in a general way. Expressions are contextually equal if they have the same termination behavior in any surrounding context. This makes contextual equivalence a strong equality, since the contexts of the language have a high discrimination power. For instance, it is not necessary to add additional tests, such as checking whether evaluation of both expressions terminates with the same values, since different values can be distinguished by contexts.

▶ **Definition 2.2.** Let $D = (\mathcal{E}, \mathcal{C}, \to, \mathcal{A}, \mathcal{T})$ be a TDPC. An expression $s \in \mathcal{E}$ *converges* if there exists $v \in \mathcal{A}$ such that $s \xrightarrow{*}_D v$. We then write $s{\downarrow_D}v$, or just $s{\downarrow_D}$ if the value v is not of interest. If $s{\downarrow_D}$ does not hold, then we say s *diverges* and write $s{\Uparrow_D}$. *Contextual preorder* $\leq_D$ and *contextual equivalence* $\sim_D$ are defined by:

$$\text{For } s_1, s_2 \in \mathcal{E}_T\text{:} \quad s_1 \leq_D s_2 \quad \text{iff} \quad \forall T' \in \mathcal{T}, C \in \mathcal{C}_{T,T'} : C[s_1]{\downarrow_D} \implies C[s_2]{\downarrow_D}$$
$$\text{For } s_1, s_2 \in \mathcal{E}_T\text{:} \quad s_1 \sim_D s_2 \quad \text{iff} \quad s_1 \leq_D s_2 \text{ and } s_2 \leq_D s_1$$

A *program transformation* ξ is a binary relation on D-expressions, such that for all $s_1\ \xi\ s_2$ the expressions s_1 and s_2 are of the same type. ξ is *correct* if for all expressions $s_1\ \xi\ s_2$ the equivalence $s_1 \sim_D s_2$ holds.

By straightforward arguments one can prove that contextual preorder is a precongruence, and contextual equivalence is a congruence.

▶ **Definition 2.3.** Let $D = (\mathcal{E}, \mathcal{C}, \to_D, \mathcal{A}, \mathcal{T})$ and $D' = (\mathcal{E}', \mathcal{C}', \to_{D'}, \mathcal{A}', \mathcal{T}')$ be TDPCs. A *translation* $\zeta : D \to D'$ consists of mappings $\zeta : \mathcal{E} \to \mathcal{E}'$, $\zeta : \mathcal{C} \to \mathcal{C}'$, such that ζ maps the identity function $\mathcal{C}$ to the identity function in $\mathcal{C}'$, and $\zeta(s)$ is closed iff s is closed.

- ζ is *convergence equivalent (ce)* if $s{\downarrow_D} \iff \zeta(s){\downarrow_{D'}}$ for all $s \in \mathcal{E}$.
- ζ is *compositional up to observation (cuo)*, if for all $C \in \mathcal{C}$ and all $s \in \mathcal{E}$ such that $C[s]$ is typed: $\zeta(C[s]){\downarrow_{D'}}$ iff $\zeta(C)[\zeta(s)]{\downarrow_{D'}}$.
- ζ is *observationally correct (oc)* if it is (ce) and (cuo).
- ζ is *adequate* if for all expressions s, t: $\zeta(s) \leq_{D'} \zeta(t) \implies s \leq_D t$.
- ζ is *fully abstract* if for all expressions s, t: $\zeta(s) \leq_{D'} \zeta(t) \iff s \leq_D t$.
- ζ is an isomorphism if ζ is fully abstract and acts as a bijection on the equivalence classes from $\mathcal{E}/\sim_D$ to $\mathcal{E}'/\sim_{D'}$.

We say D' is an *extension* of D iff $\mathcal{T} \subseteq \mathcal{T}'$, $\mathcal{E}_T \subseteq \mathcal{E}'_T$ for any type $T \in \mathcal{T}$, $\mathcal{C}_{T,T'} \subseteq \mathcal{C}'_{T,T'}$ for all types $T, T' \in \mathcal{T}$, $\mathcal{A} = \mathcal{A}' \cap \mathcal{E}$ and $\to_D \subseteq \to_{D'}$ s.t. for all $e_1 \in \mathcal{E}$ with $e_1 \to_{D'} e_2$ always $e_2 \in \mathcal{E}$ (and thus $e_1 \to_D e_2$). Given D and an extension D', the *natural embedding* of D into D' is the identity translation of $\mathcal{E}_T$ into $\mathcal{E}'_T$ and $\mathcal{C}_{T,T'}$ into $\mathcal{C}'_{T,T'}$ for all types $T, T' \in \mathcal{T}$. A natural embedding is *conservative* iff it is a fully abstract translation.

Note that a natural embedding is always convergence equivalent and compositional, which implies that it is always adequate (see [16]).

3 Untyped Lazy Lambda Calculi and Their Properties

In this section we briefly introduce four variants of the lazy lambda calculus [1] as instances of untyped TDPCs: the pure calculus AL, its extension by seq, called $\mathsf{AL}_{\mathsf{seq}}$, its extension

(β) $((\lambda x.s)\ t) \to s[t/x]$
(seq) $(\texttt{seq}\ v\ t) \to t$ if v is an answer
(case) $\texttt{case}_{K_i}\ (c_{K_i,j}\ \vec{s})\ (p_1\ \texttt{->}\ t_1) \ldots ((c_{K_i,j}\ \vec{y})\ \texttt{->}\ t_j) \ldots (p_{|K_i|}\ \texttt{->}\ t_{|K_i|}) \to t_j[\vec{s}/\vec{y}]$
(fix) $(\texttt{Fix}\ s) \to s\ (\texttt{Fix}\ s)$

Figure 1 Call-by-name reduction rules.

by data constructors and `case`, called $\mathsf{AL}_{\mathsf{cc}}$, and finally its extension by `seq` as well as data constructors and `case`, called $\mathsf{AL}_{\mathsf{cc,seq}}$.

▶ **Definition 3.1** (Lazy Lambda Calculus AL)**.** AL is the (untyped) *lazy lambda calculus* [1]. We define the components of AL according to Definition 2.1.

Expressions $\mathcal{E}$ are the set of expressions of the usual (untyped) lambda calculus, defined by the grammar $r, s, t \in \mathcal{L}_{\mathsf{AL}} ::= x \mid (s\ t) \mid \lambda x.s$. We identify α-equivalent expressions as syntactically equal according to Definition 2.1. The only reduction rule is β*-reduction* (see Fig. 1). An AL-*context* is defined as an expression in which one subexpression is replaced by the context hole $[\cdot]$. AL-*reduction contexts* R are defined by the grammar $R := [\cdot] \mid (R\ s)$, and the standard reduction in the sense of Definition 2.1 is the normal order reduction $\to_{\mathsf{AL}}$ which applies beta-reduction in a reduction context, i.e. $R[(\lambda x.s)\ t] \to_{\mathsf{AL}} R[s[t/x]]$. The answers $\mathcal{A}$ are all (also open) abstractions, which are also called *weak head normal forms (WHNF)*.

▶ **Definition 3.2** ($\mathsf{AL}_{\mathsf{seq}}$)**.** $\mathsf{AL}_{\mathsf{seq}}$ is the lazy lambda calculus extended by `seq`, i.e. expressions are defined by $r, s, t \in \mathcal{L}_{\mathsf{AL}_{\mathsf{seq}}} ::= x \mid (s\ t) \mid \lambda x.s \mid \texttt{seq}\ s\ t$. Answers are all abstractions (WHNFs). $\mathsf{AL}_{\mathsf{seq}}$-reduction contexts R are defined by the grammar $R := [\cdot] \mid (R\ s) \mid \texttt{seq}\ R\ t$, and a normal order reduction is $R[s] \to_{\mathsf{AL}_{\mathsf{seq}}} R[t]$, whenever $s \xrightarrow{\beta} t$ or $s \xrightarrow{seq} t$ (see Fig. 1).

▶ **Definition 3.3** ($\mathsf{AL}_{\mathsf{cc}}$)**.** $\mathsf{AL}_{\mathsf{cc}}$ extends AL by `case` and data constructors. There is a finite nonempty set of type constructors $K_1, \ldots, K_n$, where for every K_i there are pairwise disjoint finite nonempty sets of data constructors $\{c_{K_i,1}, \ldots c_{K_i,|K_i|}\}$. Every constructor has a fixed *arity* (a non-negative integer) denoted by $ar(K_i)$ or $ar(c_{K_i,j})$, resp. Examples are a type constructor *Bool* (of arity 0) with data constructors `True` and `False` (both of arity 0), as well as lists with a type constructor *List* (of arity 1) and data constructors `Nil` (of arity 0) and `Cons` (of arity 2). For the *constructor application* $(c_{K_i,j}\ s_1 \ldots s_{ar(c_{K_i,j})})$, we use $(c_{K_i,j}\ \vec{s})$ as an abbreviation, and write $t[\vec{s}/\vec{x}]$ for the parallel substitution $t[s_1/x_1, \ldots, s_{ar(c_{K_i,j})}/x_{ar(c_{K_i,j})}]$. The grammar $r, s, t \in \mathcal{L}_{\mathsf{AL}_{\mathsf{cc}}} ::= x \mid (s\ t) \mid \lambda x.s \mid (c_{K_i,j}\ \vec{s}) \mid (\texttt{case}_{K_i}\ s\ (c_{K_i,1}\ \vec{x}\ \texttt{->}\ s_{i,1}) \ldots (c_{K_i,|K_i|}\ \vec{x}\ \texttt{->}\ s_{i,|K_i|}))$ defines expressions of $\mathsf{AL}_{\mathsf{cc}}$. We use an abbreviation $\texttt{case}_K\ s\ alts$ if the alternatives of the `case` do not matter. The $\mathsf{AL}_{\mathsf{cc}}$-reduction contexts R are defined as $R := [\cdot] \mid (R\ s) \mid \texttt{case}_{K_i}\ R\ alts$. A normal order reduction is $R[s] \to_{\mathsf{AL}_{\mathsf{cc}}} R[t]$, where $s \xrightarrow{\beta} t$ or $s \xrightarrow{case} t$ (see Fig. 1, where p_i mean patterns $(c_{K_k,i}\ \vec{x})$ in `case`-expressions). Answers in $\mathsf{AL}_{\mathsf{cc}}$ are $\lambda x.s$ and $(c_{K_i}\ \vec{s})$, also called WHNFs.

▶ **Definition 3.4** ($\mathsf{AL}_{\mathsf{cc,seq}}$)**.** The calculus $\mathsf{AL}_{\mathsf{cc,seq}}$ combines the syntax and reduction rules of $\mathsf{AL}_{\mathsf{seq}}$ and $\mathsf{AL}_{\mathsf{cc}}$ with the obvious notion of normal order reduction $\to_{\mathsf{AL}_{\mathsf{cc,seq}}}$ applying (β), (seq), and (case) (see Fig. 1) in reduction contexts.

We will write $\lambda x_1, x_2, \ldots, x_n.t$ instead of $\lambda x_1.\lambda x_2. \ldots \lambda x_n.t$. We use the following abbreviations for specific closed lambda expressions:

$$id = \lambda x.x \qquad \omega = \lambda x.(x\ x) \qquad \Omega = (\omega\ \omega)$$
$$Y = \lambda f.\big((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))\big) \qquad \top = (Y\ (\lambda x, y.x))$$

$$
\begin{array}{ll}
\text{(caseapp)} & ((\texttt{case}_K\ t_0\ (p_1\ \texttt{->}\ t_1)\dots(p_n\ \texttt{->}\ t_n))\ r) \\
& \to (\texttt{case}_K\ t_0\ (p_1\ \texttt{->}\ (t_1\ r))\dots(p_n\ \texttt{->}\ (t_n\ r))) \\
\text{(casecase)} & (\texttt{case}_K\ (\texttt{case}_{K'}\ t_0\ (p_1\ \texttt{->}\ t_1)\dots(p_n\ \texttt{->}\ t_n))\ \ (q_1\ \texttt{->}\ r_1)\dots(q_m\ \texttt{->}\ r_m)) \\
& \qquad\qquad \to (\texttt{case}_{K'}\ t_0\ (p_1\ \texttt{->}\ (\texttt{case}_K\ t_1\ (q_1\ \texttt{->}\ r_1)\dots(q_m\ \texttt{->}\ r_m))) \\
& \qquad\qquad\qquad\qquad \dots \\
& \qquad\qquad\qquad\qquad (p_n\ \texttt{->}\ (\texttt{case}_K\ t_n(q_1\ \texttt{->}\ r_1)\dots(q_m\ \texttt{->}\ r_m)))) \\
\text{(seqseq)} & (\texttt{seq}\ (\texttt{seq}\ s_1\ s_2)\ s_3) \to (\texttt{seq}\ s_1\ (\texttt{seq}\ s_2\ s_3)) \\
\text{(seqapp)} & ((\texttt{seq}\ s_1\ s_2)\ s_3) \to (\texttt{seq}\ s_1\ (s_2\ s_3)) \\
\text{(seqcase)} & (\texttt{seq}\ (\texttt{case}_K\ t_0\ (p_1\ \texttt{->}\ t_1)\dots(p_n\ \texttt{->}\ t_n))\ r) \\
& \to (\texttt{case}_K\ t_0\ (p_1\ \texttt{->}\ (\texttt{seq}\ t_1\ r))\dots(p_n\ \texttt{->}\ (\texttt{seq}\ t_n\ r))) \\
\text{(caseseq)} & (\texttt{case}_K\ (\texttt{seq}\ s_1\ s_2)\ \mathit{alts}) \to (\texttt{seq}\ s_1\ (\texttt{case}_K\ s_2\ \mathit{alts}))
\end{array}
$$

Figure 2 `case`- and `seq`-simplifications.

It is not too hard to show that all closed diverging expressions are contextually equal. Thus we will use the symbol $\bot$ to denote a representative of the equivalence class of closed diverging expressions, e.g. one such expression is Ω.

▶ Remark. Note that contextual equivalence in all our calculi always distinguishes different values. For instance, different constructors can always be distinguished by choosing `case`-expressions as contexts such that one constructor is mapped to a value while the other one is mapped to Ω. Different abstractions are distinguished by applying them to arguments. Different variables x, y are always contextually different: The context $C := (\lambda x, y.[\cdot])\ id\ \Omega$ distinguishes them, since $C[x]$ converges, while $C[y]$ diverges.

We now show correctness of program transformations. The *simplifications* for the calculi $\mathsf{AL_{seq}}, \mathsf{AL_{cc}}, \mathsf{AL_{cc,seq}}$ are defined in Fig. 2, s.t. each simplification is defined in all calculi where the constructs exist. In [15] we prove:

▶ **Theorem 3.5.** *For $D \in \{\mathsf{AL}, \mathsf{AL_{seq}}, \mathsf{AL_{cc}}, \mathsf{AL_{cc,seq}}\}$ the reductions of the corresponding calculus (Fig. 1) and the simplifications (Fig. 2) are correct program transformations, regardless of the context they are applied in.*

Contextual equivalence of open expressions can be proven by closing them using additional lambda binders. One direction of the following lemma is obvious, since $\sim_D$ is a congruence. The other direction can be proven by using applicative bisimilarity (see [15]).

▶ **Lemma 3.6.** *For $D \in \{\mathsf{AL}, \mathsf{AL_{seq}}, \mathsf{AL_{cc}}, \mathsf{AL_{cc,seq}}\}$ and D-expressions s, t with $FV(s) \cup FV(t) \subseteq \{x_1, \dots, x_n\}$: $s \sim_D t \iff \lambda x_1, \dots, x_n.s \sim_D \lambda x_1, \dots, x_n.t$.*

Correctness of β-reduction implies that a restricted use of η-expansion is correct:

▶ **Proposition 3.7.** *For every $D \in \{\mathsf{AL}, \mathsf{AL_{seq}}, \mathsf{AL_{cc}}, \mathsf{AL_{cc,seq}}\}$ the transformation η is correct for all abstractions, i.e. $s \sim_D \lambda z.s\ z$, if s is an abstraction.*

▶ **Definition 3.8.** We use B_k^m as an abbreviation for a "bot-alternative" of the k^{th} data constructor of type constructor K_m i.e. $B_k^m := (c_{K_m,k}\ \vec{x}\ \texttt{->}\ \bot)$. Let v be any closed abstraction (for $\mathsf{AL}, \mathsf{AL_{seq}}$) or be any closed abstraction or constructor application $(c_{K_m,j}\ \vec{s})$ (for in $\mathsf{AL_{cc}}, \mathsf{AL_{cc,seq}}$), respectively.

Approximation contexts AP_i ($i \in \mathbb{N}_0$) are defined for $\mathsf{AL}, \mathsf{AL_{seq}}, \mathsf{AL_{cc}}, \mathsf{AL_{cc,seq}}$ as follows:

For AL, $\mathsf{AL_{seq}}$: $AP_0 ::= [\cdot]$ $\quad AP_{i+1} ::= (AP_i\ v) \mid (AP_i\ \bot)$

For $\mathsf{AL_{cc}}$, $\mathsf{AL_{cc,seq}}$: $AP_0 ::= [\cdot]$ $\quad AP_{i+1} ::= (AP_i\ v) \mid (AP_i\ \bot)$
$\mid \texttt{case}_{K_m}\ AP_i\ B_1^m \dots B_{j-1}^m\ (c_{K_m,j}\ \vec{x}\ \texttt{->}\ x_k)\ B_{j+1}^m \dots B_n^m$

$$s_1 := \lambda x.x\ (\lambda y.x\ \top\ \bot\ y)\ \top \qquad s_2 := \lambda x.x\ (x\ \top\ \bot)\ \top$$
$$t_1(s) := \lambda x.x\ (x\ s) \qquad t_2(s) := \lambda x.x\ \lambda z.x\ s\ z$$
where s is an expression with $FV(s) \subseteq \{x\}$
$$s_3 := \lambda x, y.x\ (y\ (y\ (x\ id))) \qquad s_4 := \lambda x, y.x\ (y\ \lambda z.y\ (x\ id)\ z)$$
$$s_5 := \lambda x, y.(x\ (x\ y))\ (x\ (x\ y)) \qquad s_6 := \lambda x, y.((x\ (x\ y))\ (x\ \lambda z.x\ y\ z))$$
$$s_7 := \lambda x.\mathtt{case}_{Bool}\ (x\ \bot)\ (\mathtt{True}\ \mathtt{->}\ \mathtt{True})\ (\mathtt{False}\ \mathtt{->}\ \bot)$$
$$s_8 := \lambda x.\mathtt{case}_{Bool}\ (x\ \lambda y.\bot)\ (\mathtt{True}\ \mathtt{->}\ \mathtt{True})\ (\mathtt{False}\ \mathtt{->}\ \bot)$$

Figure 3 The untyped counterexample expressions.

The following result known as a context lemma is proven in the technical report [15]. However, we outline the ideas of the proof: Howe's method [6, 7] implies that contextual approximation coincides with applicative similarity in all four calculi. Applicative similarity (in $\mathsf{AL}, \mathsf{AL}_{\mathsf{seq}}$) means that s_2 can simulate s_1 if and only if in case s_1 reduces to an abstraction v_1, then s_2 reduces to an abstraction v_2 and for every argument r: $v_2\ r$ can simulate $v_1\ r$. This recursive definition is meant to be co-inductive. For deriving the context lemma below, two more steps are necessary: First show that, instead of applying v_i to argument r, the definition is unchanged, if s_i is applied to argument r and reduction is then performed for $s_i\ r$. The second step is to show that the co-inductive definition is equivalent to an inductive definition, using Kleene's fixpoint theorem.

▶ **Theorem 3.9** (AP_i-Context-Lemma). *For $D \in \{\mathsf{AL}, \mathsf{AL}_{\mathsf{seq}}, \mathsf{AL}_{\mathsf{cc}}, \mathsf{AL}_{\mathsf{cc,seq}}\}$ and closed D-expressions s, t holds:*

$s \leq_D t$ iff for all i and all approximation contexts AP_i: $AP_i[s]{\downarrow_D} \implies AP_i[t]{\downarrow_D}$

We provide a criterion to prove contextual equivalence of expressions, which is used in later sections. Its proof can be found in [15].

▶ **Theorem 3.10.** *For $D \in \{\mathsf{AL}, \mathsf{AL}_{\mathsf{seq}}, \mathsf{AL}_{\mathsf{cc}}, \mathsf{AL}_{\mathsf{cc,seq}}\}$ closed D-expressions s and t are contextually equivalent if there exists $i \in \mathbb{N}_0$ such that*

1. *$AP_j[s]{\downarrow_D} \iff AP_j[t]{\downarrow_D}$ for all $0 \leq j < i$ and all AP_j-contexts.*
2. *$AP_i[s] \sim_D AP_i[t]$ for all AP_i-contexts.*

For all four calculi *applicative contexts* are defined by $A ::= [\cdot] \mid (A\ s)$. The following proposition allows systematic case-distinctions for expressions (proved in [15]).

▶ **Proposition 3.11.** *Let $D \in \{\mathsf{AL}, \mathsf{AL}_{\mathsf{seq}}, \mathsf{AL}_{\mathsf{cc}}, \mathsf{AL}_{\mathsf{cc,seq}}\}$. For every D-expression s one of the following equations holds: **1.** $s \sim_D \bot$; **2.** $s \sim_D v$ where v is an answer; **3.** $s \sim_D A[x]$ where x is a free variable and A is an applicative D-context; **4.** $s \sim_D \mathsf{seq}\ A[x]\ t'$ where x is a free variable and A is an applicative D-context, for $D \in \{\mathsf{AL}_{\mathsf{seq}}, \mathsf{AL}_{\mathsf{cc,seq}}\}$; or **5.** $s \sim_D \mathtt{case}_K\ A[x]\ alts$ where x is a free variable and A is an applicative D-context, for $D \in \{\mathsf{AL}_{\mathsf{cc}}, \mathsf{AL}_{\mathsf{cc,seq}}\}$.*

4 Relations between the Untyped Calculi

This section shows the non-conservativity of embeddings of the four untyped lazy calculi. These negative results show that the syntactically less expressive calculi are not sufficiently expressive and are thus unstable under extensions. Expressions used in our counterexamples are defined in Fig. 3. We also prove equations necessary for the examples:

▶ **Lemma 4.1.** *For all expressions s: $\top \sim_{\mathsf{AL}} \lambda x.\top$ and $\top\ s \sim_{\mathsf{AL}} \top$.*

Proof. $\top\ s\ \sim_{\mathsf{AL}}\ \top$ follows from correctness of β (Theorem 3.5), since $\top\ s \xrightarrow{\beta,*} \lambda x.\lambda z.(\lambda x.\lambda z.(x\ x))\ (\lambda x.\lambda z.(x\ x)) \xleftarrow{\beta,*} \top$. For $\top \sim_{\mathsf{AL}} \lambda x.\top$ we use Theorem 3.10 (for $i=1$): $\top{\downarrow_D}, \lambda x.\top{\downarrow_D}$, and $(\lambda x.\top)\ r \xrightarrow{\beta} \top \sim_D (\top\ r)$ for any r. ◀

▶ **Theorem 4.2.** *The following equalities hold for the expressions in Fig. 3:* ***1.*** *If $s[id/x] \not\sim_{\mathsf{AL}} \bot$ then $t_1(s) \sim_{\mathsf{AL}} t_2(s)$.* ***2.*** *$s_1 \sim_{\mathsf{AL}} s_2$.* ***3.*** *$s_3 \sim_{\mathsf{AL}} s_4$.* ***4.*** *$s_5 \sim_{\mathsf{AL}_{\mathsf{seq}}} s_6$.* ***5.*** *$s_7 \sim_{\mathsf{AL}_{\mathsf{cc}}} s_8$.*

Proof. **1.** We use Theorem 3.10 (for $i=1$). For the empty context we have $t_1(s){\downarrow_{\mathsf{AL}}}$ and $t_2(s){\downarrow_{\mathsf{AL}}}$. Now we consider the case $(t_1\ b)$ and $(t_2\ b)$ where b is a closed abstraction or $\bot$. We make a case distinction on the argument b according to Proposition 3.11. By easy computations $(t_1\ b) \sim_{\mathsf{AL}} (t_2\ b)$ if $b=\bot$, $b=\lambda x.\bot$, or $b=\lambda x_1.\lambda x_2.t$. For $b := \lambda x.x$, two β-reductions show that $t_1\ \lambda x.x \sim_{\mathsf{AL}} s[id/x]$, and that $t_2\ \lambda x.x \sim_{\mathsf{AL}} \lambda z.s[id/x]\ z$. Since $s[id/x] \not\sim_{\mathsf{AL}} \bot$, it is equivalent to an abstraction, and Proposition 3.7 shows contextual equivalence of the two expressions. Now let $b := \lambda u.u\ t_1\ \ldots\ t_n$ with $n \geq 1$. If $(b\ s[b/x]) \not\sim_{\mathsf{AL}} \bot$, then there exists a closed abstraction $\lambda w.s'$ such that $(\lambda w.s') \sim_{\mathsf{AL}} (b\ s[b/x])$. By Proposition 3.7 we can transform: $(t_1\ b) \sim_{\mathsf{AL}} b\ (b\ s[b/x]) \sim_{\mathsf{AL}} b\ \lambda w.s' \sim_{\mathsf{AL}} b\ \lambda z.(\lambda w.s')\ z \sim_{\mathsf{AL}} b\ \lambda z.(b\ s[b/x]\ z) \sim_{\mathsf{AL}} t_2\ b$. In the case $(b\ s[b/x]) \sim_{\mathsf{AL}} \bot$, evaluation of $(\lambda u.u\ t_1 \ldots t_n)\ \bot$ and $(\lambda u.u\ t_1 \ldots t_n)\ (\lambda y.\bot)$ results in $\bot$.

2. We use Theorem 3.10 (for $i=1$). Since $s_1{\downarrow_{\mathsf{AL}}}$ and $s_2{\downarrow_{\mathsf{AL}}}$, we only consider the cases $(s_1\ b)$ and $(s_2\ b)$ where b is a closed abstraction or $\bot$. We use Proposition 3.11 for a case distinction on b. It is easy to verify that $s_1\ b \sim_{\mathsf{AL}} s_2\ b$ for $b=\bot$, $b=\lambda z.\bot$, and $b=\lambda z.z$. For $b := \lambda z.(z\ u_1 \ldots u_n)$ where $n \geq 1$, we have $(s_1\ b) \sim_{\mathsf{AL}} b\ (\lambda y.\top)\ \top$ and $(s_2\ b) \sim_{\mathsf{AL}} b\ \top\ \top$, and by Lemma 4.1 also $(s_1\ b) \sim_{\mathsf{AL}} (s_2\ b)$.

3. We use Theorem 3.10 (for $i=2$). Since $s_j{\downarrow_{\mathsf{AL}}}$ and $(s_j\ b){\downarrow_{\mathsf{AL}}}$ for $j=3,4$ we need to consider the cases $(s_3\ b\ d)$ and $(s_4\ b\ d)$ where b,d are closed abstractions or $\bot$. We use Proposition 3.11 for case distinction on d. If $d=\bot$, or $d=\lambda x.\bot$, then $s_3\ b\ d \sim_{\mathsf{AL}} s_4\ b\ d$. If $d := \lambda x.x$, then item 1 shows that $\lambda x.x\ (x\ id) \sim_{\mathsf{AL}} \lambda x.x\ \lambda z.(x\ id)\ z$. Correctness of β implies that $b\ (b\ id) \sim_{\mathsf{AL}} b\ \lambda z.(b\ id)\ z$, and thus $s_3\ b\ d \sim_{\mathsf{AL}} b\ (b\ id) \sim_{\mathsf{AL}} b\ \lambda z.(b\ id)\ z \sim_{\mathsf{AL}} s_4\ b\ d$. If $d := \lambda x_1.\lambda x_2.t$, then $s_3\ b\ d \sim_{\mathsf{AL}} b\ (d\ (\lambda x_2.t[(b\ id)/x_1])) \xrightarrow{\eta} b\ (d\ \lambda.z\ (\lambda x_2.t[(b\ id)/x_1])\ z) \sim_{\mathsf{AL}} s_4\ b\ d$, where η is correct by Proposition 3.7. If $d := \lambda u.u\ t_1\ \ldots\ t_n$ with $n \geq 1$ and $(d\ (b\ id)) \not\sim_{\mathsf{AL}} \bot$, it is equivalent to an abstraction, and η is correct, hence equivalence holds in this case. Otherwise, if $(d\ (b\ id)) \sim_{\mathsf{AL}} \bot$, then $(b\ (d\ \bot) \sim_{\mathsf{AL}} (b\ (d\ \lambda x.\bot))$ since $(d\ \bot) \sim_{\mathsf{AL}} \bot \sim_{\mathsf{AL}} (d\ \lambda x.\bot)$.

4. We use Theorem 3.10 (for $i=2$). We have $s_j{\downarrow_{\mathsf{AL}_{\mathsf{seq}}}}$ and $(s_j\ b){\downarrow_{\mathsf{AL}_{\mathsf{seq}}}}$ for any b for $j=5,6$. Now we consider the cases $(s_5\ b\ d)$ and $(s_6\ b\ d)$ where b,d are closed abstractions or $\bot$. We make a case distinction on b using Proposition 3.11. The cases $b=\bot$, $b=\lambda x.\bot$, and $b=\lambda u.u$ are easy to verify. If $b=\lambda u,v.b'$ then the subexpression $(b\ d)$ is contextually equivalent to $\lambda v.b'[d/u]$ Thus, η-expansion for $(b\ d)$ is correct which shows $s_5\ b\ d\ \sim_{\mathsf{AL}_{\mathsf{seq}}} s_6\ b\ d$. For the other case we distinguish whether $(b\ d) \sim_{\mathsf{AL}_{\mathsf{seq}}} \bot$ holds. If $(b\ d) \not\sim_{\mathsf{AL}_{\mathsf{seq}}} \bot$ then η is correct, which shows that $s_5\ b\ d\ \sim_{\mathsf{AL}_{\mathsf{seq}}} s_6\ b\ d$. If $(b\ d) \sim_{\mathsf{AL}_{\mathsf{seq}}} \bot$, then we have to check more cases: If $b=\lambda u.\mathtt{seq}\ (u\ t_1 \ldots)\ r$ or $b=\lambda u.\mathtt{seq}\ u\ r$, then $(b\ (b\ d)) \sim_{\mathsf{AL}_{\mathsf{seq}}} \bot$, and $s_5\ b\ d$ is equivalent to $s_6\ b\ d$. If $b=\lambda u.u\ t_1 \ldots$, then $(b\ (b\ d))$ becomes $\bot$ in both expressions, which shows $(s_5\ b\ d) \sim_{\mathsf{AL}_{\mathsf{seq}}} \bot \sim_{\mathsf{AL}_{\mathsf{seq}}} (s_6\ b\ d)$.

5. We use Theorem 3.10. Since $s_7{\downarrow_{\mathsf{AL}_{\mathsf{cc}}}}$, $s_8{\downarrow_{\mathsf{AL}_{\mathsf{cc}}}}$, and $\mathtt{case}\ s_7\ \ldots \sim_{\mathsf{AL}_{\mathsf{cc}}} \bot \sim_{\mathsf{AL}_{\mathsf{cc}}} \mathtt{case}\ s_8\ \ldots$, it is sufficient to show $(s_7\ b)\ \sim_{\mathsf{AL}_{\mathsf{cc}}} (s_8\ b)$ where b is a closed abstraction, a constructor application, or $\bot$. If $b=\bot$ then the equivalence holds. Otherwise, we inspect the cases of a normal-order reduction for $b\ y$ for some free variable y: If $b\ y{\downarrow_{\mathsf{AL}_{\mathsf{cc}}}}\mathtt{True}$ (or $b\ y{\downarrow_{\mathsf{AL}_{\mathsf{cc}}}}\mathtt{False}$, resp.) then $(b\ \bot)$ and $(b\ \lambda x.\bot)$ also converge with $\mathtt{True}$ (or $\mathtt{False}$, resp.), which shows that $(s_7\ b)\ \sim_{\mathsf{AL}_{\mathsf{cc}}}\ (s_8\ b)$. If evaluation of $b\ y$ stops with $R[y]$ for some

reduction context R, then $(s_7\ b)$ evaluates to $\mathtt{case}_{Bool}\ R[\bot]\ alts$ which is equivalent to $\bot$, and $(s_8\ b)$ evaluates to $\mathtt{case}_{Bool}\ R[\lambda x.\bot]\ alts$. We consider cases of R: If $R = [\cdot]$ then $(s_8\ b) \sim_{\mathsf{AL_{cc}}} \bot$. If $R = R'[[\cdot]\ r]$ then $(s_8\ b)$ evaluates to $\mathtt{case}_{Bool}\ R'[\bot]\ alts$ which is equivalent to $\bot$. Finally, if $R = R'[\mathtt{case}_K\ [\cdot]\ alts']$ then $(s_8\ b) \sim_{\mathsf{AL_{cc}}} \bot$. If $(b\ y)$ converges with $c_{K_i,j}\ t_1\ \ldots\ t_n$ for some constructor $c_{K_i,k}$ not of type $Bool$ then $(b\ \bot)$ converges to $c_{K_i,j}\ t'_1\ \ldots\ t'_n$ and $(b\ \lambda x.\bot)$ converges to $c_{K_i,j}\ t''_1\ \ldots\ t''_n$. However, in this case $(s_7\ b) \sim_{\mathsf{AL_{cc}}} \bot \sim_{\mathsf{AL_{cc}}} (s_8\ b)$. ◀

Now we obtain non-conservativity for all embeddings between the four calculi as follows:

▶ **Theorem 4.3.** *The natural embeddings of* AL *in* $\mathsf{AL_{seq}}$, AL *in* $\mathsf{AL_{cc,seq}}$, AL *in* $\mathsf{AL_{cc}}$, $\mathsf{AL_{seq}}$ *in* $\mathsf{AL_{cc,seq}}$, *and* $\mathsf{AL_{cc}}$ *in* $\mathsf{AL_{cc,seq}}$ *are not conservative.*

Proof. AL in $\mathsf{AL_{seq}}$ and AL in $\mathsf{AL_{cc,seq}}$: The proof uses the expressions s_1, s_2 which are adapted from the example of [4, Proposition 3.15]. Theorem 4.2, item 2 shows that $s_1 \sim_{\mathsf{AL}} s_2$. The context $C := ([\cdot]\ \lambda z.\mathtt{seq}\ z\ id)$ distinguishes s_1, s_2 in $\mathsf{AL_{seq}}, \mathsf{AL_{cc,seq}}$, since $C[s_1]{\downarrow_D}$ while $C[s_2]{\Uparrow_D}$ for $D \in \{\mathsf{AL_{seq}}, \mathsf{AL_{cc,seq}}\}$.
Another counterexample uses the expressions $t_1(s), t_2(s)$ with $s = (x\ ((x\ id)\ (x\ id)))$: Since $s[id/x] \sim_{\mathsf{AL}} id$, Theorem 4.2, item 1 shows $t_1(s) \sim_{\mathsf{AL}} t_2(s)$. However, the context $C := ([\cdot]\ \lambda y.\ \mathtt{seq}\ y\ \omega)$ distinguishes $t_1(s)$ and $t_2(s)$ in $\mathsf{AL_{seq}}$ and $\mathsf{AL_{cc,seq}}$.

AL in $\mathsf{AL_{cc}}$: From Theorem 4.2 we have $s_3 \sim_{\mathsf{AL}} s_4$. In $\mathsf{AL_{cc}}$ the context $C := [\cdot]\ (\lambda u.u\ \mathtt{True})\ (\lambda u.\mathtt{case}_{Bool}\ u\ (\mathtt{True}\ \mathtt{->}\ \mathtt{False})\ (\mathtt{False}\ \mathtt{->}\ id))$ distinguishes s_3 and s_4, since $C[s_3] \sim_{\mathsf{AL_{cc}}} \mathtt{True}$ and $C[s_4]{\Uparrow_{\mathsf{AL_{cc}}}}$.

$\mathsf{AL_{seq}}$ in $\mathsf{AL_{cc,seq}}$: Theorem 4.2 shows $s_5 \sim_{\mathsf{AL_{seq}}} s_6$. In $\mathsf{AL_{cc,seq}}$ the context $C := ([\cdot]\ b\ \mathtt{True})$ with $b := \lambda u.\mathtt{case}_{Bool}\ u\ (\mathtt{True}\ \mathtt{->}\ \mathtt{False})(\mathtt{False}\ \mathtt{->}\ id)$ distinguishes s_5 and s_6, since $C[s_5] \xrightarrow{*}_{\mathsf{AL_{cc,seq}}} id$, but $C[s_6]{\Uparrow_{\mathsf{AL_{cc,seq}}}}$.

$\mathsf{AL_{cc}}$ in $\mathsf{AL_{cc,seq}}$: A counterexample for conservativity of embedding $\mathsf{AL_{cc}}$ into $\mathsf{AL_{cc,seq}}$ was given in [18] which can be translated into the notations of this paper as follows: The equation $s_7 \sim_{\mathsf{AL_{cc}}} s_8$ holds (Theorem 4.2), but for the context $C := [\cdot]\ \lambda u.\mathtt{seq}\ u\ \mathtt{True}$ we have $C[s_8]{\downarrow_{\mathsf{AL_{cc,seq}}}}\mathtt{True}$ while $C[s_7]{\Uparrow_{\mathsf{AL_{cc,seq}}}}$. ◀

5 Monomorphically Typed Calculi and Embeddings

We now analyze embeddings among the four calculi under monomorphic typing, and therefore we add a monomorphic type system to the calculi. The counterexamples in Sect. 4 cannot be transferred to the typed calculi except for the counterexample showing non-conservativity of embedding $\mathsf{AL_{cc}}$ into $\mathsf{AL_{cc,seq}}$.

Since AL with a monomorphic type system is the simply typed lambda calculus (which is too inexpressive since every expression converges) we extend all the calculi by a fixpoint combinator `Fix` as a constant to implement recursion, and by a constant `Bot` to denote a diverging expression[1]. The resulting calculi are called AL^τ, $\mathsf{AL}^\tau_{\mathsf{seq}}$, $\mathsf{AL}^\tau_{\mathsf{cc}}$, $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$.

The syntax for types is $T ::= o \mid T \to T \mid K(T_1, \ldots, T_{arK})$, where o is the base type, and K is a type constructor. The syntax for expressions is as in the base calculi, but extended by `Fix` as a family of constants of all types of the form $(T \to T) \to T$, and the constant `Bot` as a family of constants of all types. Variables have a built-in type, i.e. in an expression every variable is annotated with a monomorphic type, e.g. $\lambda x^{o\to o}.x^{o\to o}$ is an identity on functions of type $o \to o$. However, we rarely write these annotations explicitly. The type of constructors

[1] `Bot` can also be encoded using `Fix`, but for convenient representation we include the constant.

(botapp) $(\texttt{Bot}\ s) \to \texttt{Bot}$ (botseq) $(\texttt{seq}\ \texttt{Bot}\ s) \to \texttt{Bot}$
(botcase) $(\texttt{case}_K\ \texttt{Bot}\ alts) \to \texttt{Bot}$

Figure 4 The bot-simplifications.

is structured as in a polymorphic calculus: The family of constructors for one constructor c_{K_i} has a (polymorphic) type schema of the form $T_1 \to \ldots \to T_n \to (K_i\ T'_1\ \ldots\ T'_{ar(K_i)})$, where every type-variable of $T_1 \to \ldots \to T_n$ is contained in $(K_i\ T'_1\ \ldots\ T'_{ar(K_i)})$, and every monomorphic type of constructor c_{K_i} is an instance of this type. The types of `case` and `seq` are the monomorphic instances of the usual polymorphic types as in Haskell. We omit the standard typing rules. However, we write $s :: T$ which means that s can be typed by a (monomorphic) type T. The reduction rules are in Fig. 1, and normal order reduction $\to_D$ for $D \in \{\mathsf{AL}^\tau, \mathsf{AL}^\tau_{\mathsf{seq}}, \mathsf{AL}^\tau_{\mathsf{cc}}, \mathsf{AL}^\tau_{\mathsf{cc,seq}}\}$ applies the reduction rules in reduction contexts (defined as before). It is easy to verify that normal order reduction is deterministic, type-, and closedness-preserving. The following progress lemma holds: for every closed expression t, either $t \xrightarrow{*}_D t_0$, where t_0 is a value, or t has an infinite reduction sequence, or $t \xrightarrow{*}_D R[\texttt{Bot}]$, where R is a reduction context. In particular, the typing implies that case-expressions $(\texttt{case}_K\ (c\ \ldots)\ alts)$ are always reducible by a case-reduction.

Answers are defined as abstractions, constructor applications, and the constant `Fix`. Contextual equivalence $\sim_D$ is defined according to Definition 2.2. We also reuse the approximation contexts, but restrict them to well-typed contexts. The AP_i-context lemma (Theorem 3.9) also holds for the typed calculi, where only equally typed expressions and well-typed contexts are taken into account.

▶ **Theorem 5.1.** *For $D \in \{\mathsf{AL}^\tau, \mathsf{AL}^\tau_{\mathsf{seq}}, \mathsf{AL}^\tau_{\mathsf{cc}}, \mathsf{AL}^\tau_{\mathsf{cc,seq}}\}$ and closed, equally typed D-expressions s, t holds: $s \le_D t$ iff for all i and all approximation contexts AP_i, such that $AP_i[s]$ and $AP_i[t]$ are well-typed: $AP_i[s]{\downarrow_D} \implies AP_i[t]{\downarrow_D}$.*

To lift the correctness results for program transformations into the typed calculi, we define a translation δ.

▶ **Definition 5.2.** Let $\delta : \mathsf{AL}^\tau_{\mathsf{cc,seq}} \to \mathsf{AL}_{\mathsf{cc,seq}}$ be the translation of an $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$-expression that first removes all types and then leaves all syntactical constructs as they are except for the cases $\delta(\texttt{Bot}) := \Omega$ and $\delta(\texttt{Fix}) := \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$.

In [15] we prove adequacy of δ, which implies that reduction rules and simplifications are correct program transformations in the typed calculi.

▶ **Proposition 5.3.** *For equally typed $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$-expressions s, t it holds: $\delta(s) \sim_{\mathsf{AL}_{\mathsf{cc,seq}}} \delta(t)$ implies $s \sim_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}} t$. The same holds for AL^τ, $\mathsf{AL}^\tau_{\mathsf{seq}}$, and $\mathsf{AL}^\tau_{\mathsf{cc}}$ w.r.t. their untyped variants.*

▶ **Theorem 5.4.** *All reduction rules and simplifications in Figs. 1, 2, and 4 are correct program transformations in AL^τ, $\mathsf{AL}^\tau_{\mathsf{cc}}$, $\mathsf{AL}^\tau_{\mathsf{seq}}$, and $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$.*

We now show non-conservativity of embedding AL^τ in $\mathsf{AL}^\tau_{\mathsf{seq}}$ as well as of $\mathsf{AL}^\tau_{\mathsf{cc}}$ in $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$, i.e. the addition of `seq` is not conservative. For the other embeddings, AL^τ in $\mathsf{AL}^\tau_{\mathsf{cc}}$ and $\mathsf{AL}^\tau_{\mathsf{seq}}$ in $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$, we show conservativity. This is consistent with typability: the counterexample for AL in $\mathsf{AL}_{\mathsf{cc}}$ requires an untyped context, and the counterexample for $\mathsf{AL}_{\mathsf{seq}}$ in $\mathsf{AL}_{\mathsf{cc,seq}}$ has a self-application of an expression, which is nontypable.

5.1 Adding Seq is Not Conservative

We consider calculi AL^{τ} and $\mathsf{AL}^{\tau}_{\mathsf{seq}}$.

There is only one equivalence class w.r.t. contextual equivalence for closed expressions of type o: it is $\mathtt{Bot}^o :: o$. For type $o \to o$, there are only two equivalence classes with representatives $\mathtt{Bot}^{o\to o}$ and $\lambda x^o.\mathtt{Bot}^o$. Note that the expression $\lambda x^o.x^o$ is equivalent to $\lambda x^o.\mathtt{Bot}^o$, since there are no values of type o.

Our counterexample to conservativity are the following expressions s_9, s_{10} of type $((o \to o) \to (o \to o) \to (o \to o)) \to (o \to o) \to (o \to o) \to (o \to o) \to (o \to o)$:

$$s_9 := \lambda f,x,y,z.f\ (f\ x\ y)\ (f\ y\ z) \qquad s_{10} := \lambda f,x,y,z.f\ (f\ x\ x)\ (f\ z\ z)$$

▶ **Theorem 5.5.** *The embedding of* AL^{τ} *into* $\mathsf{AL}^{\tau}_{\mathsf{seq}}$ *and into* $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$ *is not conservative*

Proof. We use Theorem 3.10 (with $i = 1$) which also holds in the typed calculi (see [15]) and show $s_9 \sim_{\mathsf{AL}^{\tau}} s_{10}$. Since $s_9{\downarrow_{\mathsf{AL}^{\tau}}}$ and $s_{10}{\downarrow_{\mathsf{AL}^{\tau}}}$, we need to show $s'_9 := (s_9\ b) \sim_{\mathsf{AL}^{\tau}} s'_{10} := (s_{10}\ b)$, where b is a closed expression of type $(o \to o) \to (o \to o) \to (o \to o)$. We check the different cases for b. Due to its type b must be equivalent to one of $\mathtt{Bot}$, $\lambda w.\mathtt{Bot}$, $\lambda u,w.\mathtt{Bot}$, $\lambda w_1,w_2,w_3.\mathtt{Bot}$, $\lambda x,y.x$, and $\lambda x,y.y$. For the first three cases it holds: $s'_9 \sim_{\mathsf{AL}^{\tau}} \lambda x,y,z.\mathtt{Bot} \sim_{\mathsf{AL}^{\tau}} s'_{10}$. If $b = \lambda w_1,w_2,w_3.\mathtt{Bot}$ then $s'_9 \sim_{\mathsf{AL}^{\tau}} \lambda x,y,z,w_3.\mathtt{Bot} \sim_{\mathsf{AL}^{\tau}} s'_{10}$. If $b = \lambda x,y.x$ then $s'_9 \sim_{\mathsf{AL}^{\tau}} \lambda x,y,z.x \sim_{\mathsf{AL}^{\tau}} s'_{10}$. If $b = \lambda x,y.y$ then $s'_9 \sim_{\mathsf{AL}^{\tau}} \lambda x,y,z.z \sim_{\mathsf{AL}^{\tau}} s'_{10}$. Non-conservativity now follows from the context $C = ([\cdot]\ (\lambda x,y.\mathtt{seq}\ x\ y)\ (\lambda x.\mathtt{Bot})\ \mathtt{Bot}\ (\lambda x.\mathtt{Bot}))$: The expressions $C[s_9]$, $C[s_{10}]$, are typable in $\mathsf{AL}^{\tau}_{\mathsf{seq}}, \mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$ and $C[s_9] \sim_D \mathtt{Bot}$, but $C[s_{10}] \sim_D (\lambda x.\mathtt{Bot})$ for $D \in \{\mathsf{AL}^{\tau}_{\mathsf{seq}}, \mathsf{AL}^{\tau}_{\mathsf{cc,seq}}\}$. ◀

We reuse the counterexample in the untyped case represented by expressions s_7 and s_8, where $\bot$ is replaced by $\mathtt{Bot}$. The example becomes

$$\begin{array}{lcl} s_{11} & := & \lambda x.\mathtt{case}_{Bool}\ (x\ \mathtt{Bot})\ (\mathtt{True}\ \mathtt{->}\ \mathtt{True})\ (\mathtt{False}\ \mathtt{->}\ \mathtt{Bot}) \\ s_{12} & := & \lambda x.\mathtt{case}_{Bool}\ (x\ (\lambda y.\mathtt{Bot}))\ (\mathtt{True}\ \mathtt{->}\ \mathtt{True})\ (\mathtt{False}\ \mathtt{->}\ \mathtt{Bot}) \end{array}$$

where s_{11}, s_{12} are typed as $((T \to Bool) \to Bool)$ for any type T. The two expressions are equivalent in $\mathsf{AL}^{\tau}_{\mathsf{cc}}$: They are typed, and $\delta(s_{11}) \sim_{\mathsf{AL}_{\mathsf{cc}}} \delta(s_{12})$ (see Theorem 4.2, item 5). Thus Proposition 5.3 is applicable. However, $s_{11} \not\sim_{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}} s_{12}$, since $s_{12}\ b$ evaluates to $\mathtt{True}$, while $s_{11}\ b$ diverges, where $b = \lambda u.\mathtt{seq}\ u\ \mathtt{True}$.

▶ **Theorem 5.6.** *The embedding of* $\mathsf{AL}^{\tau}_{\mathsf{cc}}$ *into* $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$ *is not conservative.*

5.2 Adding Case and Constructors is Conservative

We show that adding case and constructors to the monomorphically typed calculi is conservative. We give a detailed proof for embedding $\mathsf{AL}^{\tau}_{\mathsf{seq}}$ into $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$. The proof for embedding AL^{τ} into $\mathsf{AL}^{\tau}_{\mathsf{cc}}$ is analogous by omitting unnecessary cases. We show that for $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-expressions s,t the embedding is fully abstract, i.e. $s \le_{\mathsf{AL}^{\tau}_{\mathsf{seq}}} t \iff s \le_{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}} t$. The hard part is $s \le_{\mathsf{AL}^{\tau}_{\mathsf{seq}}} t \implies s \le_{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}} t$. Lemma 3.6 holds in the typed calculi as well, and thus it suffices to consider closed s,t. The AP_i-context lemma (Theorem 5.1) can be used, where the arguments are closed.

The main argument concerns the following situation: There are closed equally typed $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-expressions s,t, such that $s \le_{\mathsf{AL}^{\tau}_{\mathsf{seq}}} t$, but we assume that $s \le_{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}} t$ does not hold. Since s,t must have a type without constructed types and since the AP_i-context lemma holds, there is an $n \ge 0$, and $v_i, i = 1,\ldots,n$, that are $\mathtt{Bot}$ or $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$-values, and where all v_i are of an $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type, such that $s\ v_1 \ldots v_n{\downarrow_{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}}}$, but $t\ v_1 \ldots v_n{\Uparrow_{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}}}$. The goal is to show that there are $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-expressions v'_i that are $\mathtt{Bot}$ or $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-values, such that $s\ v'_1 \ldots v'_n{\downarrow_{\mathsf{AL}^{\tau}_{\mathsf{seq}}}}$, and

$t\ v'_1 \dots v'_n \Uparrow_{\mathsf{AL}^\tau_{\mathsf{seq}}}$ which refutes $s \leq_{\mathsf{AL}^\tau_{\mathsf{seq}}} t$ and thus leads to a contradiction. It is sufficient to show that for every $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$-value v and context C with $C[v]{\downarrow_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}}}$, there is an $\mathsf{AL}^\tau_{\mathsf{seq}}$-value v', with $v' \leq_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}} v$, such that $C[v']{\downarrow_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}}}$.

In order to construct the proof we define simplification transformations in our monomorphically typed calculi, whenever the appropriate constructs exist in the calculus.

▶ **Definition 5.7.** The *simplification rules* (caseapp), (casecase), (seqseq), (seqapp), (seqcase), (caseseq), (botapp), (botcase), and (botseq) are defined in Figs. 2 and 4, where we use the typed variants. For $D \in \{\mathsf{AL}^\tau_{\mathsf{cc,seq}}, \mathsf{AL}^\tau_{\mathsf{cc}}\}$ let $\xrightarrow{Dx}$ denote the reduction using normal order reductions and simplification rules in a reduction context, where in case of a conflict the topmost redex is reduced. If $s \xrightarrow{Dx,*} v$ for some D-answer v, then we denote this as $s{\downarrow_{Dx}}$.

Let $\xrightarrow{bcsfC}$ denote the reduction in any context by (β), (case), (seq), and (fix).

The simplifications are correct in the calculi under consideration and they do not change the normal order reduction length (proven in [15]):

▶ **Lemma 5.8.** *In the calculi $\mathsf{AL}^\tau_{\mathsf{cc}}$ and $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$: The simplification rules preserve the length of (converging) normal order reductions, i.e. let d be a simplification rule and $D \in \{\mathsf{AL}^\tau_{\mathsf{cc}}, \mathsf{AL}^\tau_{\mathsf{cc,seq}}\}$: if $s \xrightarrow{d} s'$ then $s \xrightarrow{n}_D v$, where v is a D-WHNF, if and only if $s' \xrightarrow{n}_D v'$, where v' is a D-WHNF.*

▶ **Lemma 5.9.** *For $D \in \{\mathsf{AL}^\tau_{\mathsf{cc,seq}}, \mathsf{AL}^\tau_{\mathsf{cc}}\}$ we have $\downarrow_D = \downarrow_{Dx}$.*

Proof. Since the simplification rules are correct in $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$, $\mathsf{AL}^\tau_{\mathsf{cc}}$, $s{\downarrow_{Dx}}$ implies that $s{\downarrow_D}$. Now assume that $s{\downarrow_D}$. We use induction on the number of (β), (case), (seq), (fix)-reductions of s to a WHNF. If s is a WHNF, then it is irreducible w.r.t. $\xrightarrow{Dx}$. If s has a normal order reduction of length $n > 0$ to a WHNF, then consider a $\xrightarrow{Dx}$-reduction sequence $s \xrightarrow{Dx,*} s_0$, where s_0 is a D-WHNF. Lemma 5.8 and termination of the simplifications (proved in [15]) show that there are s', s'', such that $s \xrightarrow{Dx,*} s' \to_D s''$, where $s \xrightarrow{Dx,*} s'$ consists only of simplification rules. Lemma 5.8 shows that the normal order reduction length of s'' to a WHNF is smaller than n. Now we can apply the induction hypothesis. ◀

▶ **Definition 5.10.** The following approximation procedure computes for every D-expression t (for $D \in \{\mathsf{AL}^\tau_{\mathsf{cc,seq}}, \mathsf{AL}^\tau_{\mathsf{cc}}\}$) and every depth i an approximating expression $approx(t,i) \leq_D t$. First a pre-approximation is computed where $preapprox(t,0) := \mathtt{Bot}$. If there is an infinite $\xrightarrow{Dx}$-reduction sequence starting with t, then $preapprox(t,i) := \mathtt{Bot}$ for all for $i > 0$. Otherwise, let $t \xrightarrow{Dx,*} t'$ where t' is irreducible for $\xrightarrow{Dx}$. Let M be the multicontext derived from t' where every subexpression at depth one is a hole, such that $t' = M(t_1, \dots, t_k)$, and t_j, for $1 \leq j \leq k$, are subexpressions at depth 1. Let $t'_j = preapprox(t_j, i-1)$ for $j = 1, \dots, k$, and define the result as $preapprox(t,i) := M(t'_1, \dots, t'_k)$.

Finally, $approx(t,i)$ is computed from $preapprox(t,i)$ by computing its normal form under the bot-simplifications in Fig. 4.

E.g. for $t = \mathtt{seq}\ (\mathtt{seq}\ x\ id)\ id$ first $t \xrightarrow{Dx,*} (\mathtt{seq}\ x\ (\mathtt{seq}\ id\ id))$. Replacing the subexpressions at depth 1 by $\mathtt{Bot}$ results in $preapprox(t,1) = (\mathtt{seq}\ \mathtt{Bot}\ \mathtt{Bot})$ which reduces to $approx(t,1) = \mathtt{Bot}$. Similarly, $preapprox(t,2) = approx(t,2) = (\mathtt{seq}\ x\ \lambda z.\mathtt{Bot})$.

▶ **Lemma 5.11.** *For $D \in \{\mathsf{AL}^\tau_{\mathsf{cc,seq}}, \mathsf{AL}^\tau_{\mathsf{cc}}\}$: $approx(t,i) \leq_D t$.*

We show a variant of the so-called subterm property for approximations:

▶ **Lemma 5.12.** *The approximations $approx(t,i)$ are of the same type as t and irreducible w.r.t. the simplification rules and $\xrightarrow{bcsfC}$-irreducible. If t is an $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$-expression of $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type, then $approx(t,i)$ is an $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-expression. If t is an $\mathsf{AL}^{\tau}_{\mathsf{cc}}$-expression of AL^{τ}-type, then $approx(t,i)$ is an AL^{τ}-expression.*

Proof. The expressions $approx(t,i)$ have the same type as t. Only bot-simplifications may be possible, and these can only enable other bot-simplifications and thus, every $approx(t,i)$ is irreducible w.r.t. the simplification rules. It remains to show that $a := approx(t,i)$ must be an $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-expression ($\mathsf{AL}^{\tau}$-expression, resp.). W.l.o.g. we consider the case with `seq`-expressions.

Suppose that there is a subexpression in $a = approx(t,i)$ of non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type. We select the subexpressions of non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type that are not contained in another subexpression of non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type; let s denote the one of a maximal non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type among these subexpressions. Since a is closed, we obtain that s cannot be a variable, since then either there is a superterm of s that is an abstraction of non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type, or a case-expression of non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type. Since a is of $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type, and s is maximal, there must be an immediate superterm s' of s which is of $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type. We look for the structure of s'. Due to the maximality conditions, s' cannot be an abstraction, an application of the form $(s_0\ s)$, a constructor application, a seq-expression of the form $(\mathtt{seq}\ s_0\ s)$, or a case-alternative, since then it would also have a non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type. It may be an application $(s\ s_2)$, a seq-expression $(\mathtt{seq}\ s\ s_2)$, or a case expression $\mathtt{case}\ s\ alts$.

First assume that s' is an application, then let s_0 be the leftmost and topmost non-application in s, i.e. $s' = (s_0\ r_1 \ldots r_n)$, and $s = (s_0\ r_1 \ldots r_{n-1})$, $n \geq 1$, where s_0 is not an application. The expression s_0 must be of non-$\mathsf{AL}^{\tau}_{\mathsf{seq}}$-type. Then s_0 cannot be `Bot`, an abstraction, `Fix`, a `case`-expression, or a `seq`-expression, since otherwise the subterm $s_0\ r_1$ would be reducible by (botapp), (β), (fix), (caseapp), or (seqapp). s_0 cannot be a constructor application either, due to types. Hence s' is not an application.

If s' is a case expression $\mathtt{case}_K\ s\ alts$, then s cannot be `Bot`, a `case`-expression, a `seq`-expression, or a constructor-application, since otherwise s would be reducible by (botcase), (casecase), (case), or (caseseq). Due to typing s cannot be an abstraction or `Fix`, and finally s' cannot be an application using the arguments above. Hence s' is not a case-expression.

If s' is a `seq`-expression $\mathtt{seq}\ s\ s_2$, then s cannot be `Bot`, an abstraction, `Fix`, a constructor application, a `case`-expression, or a `seq`-expression, since then s' would be reducible by (botseq), (seq), (seqcase), or (seqseq). s cannot be an application either, as argued above. Hence s' cannot be `seq`-expression.
In summary, such a subexpression does not exist, i.e. $approx(t,i)$ is an $\mathsf{AL}^{\tau}_{\mathsf{seq}}$-expression. ◀

In the following we use $s|_p$ for the subterm of s at position p, and $s[\cdot]_p$ for the expression s where the subterm at position p is replaced by a context hole.

▶ **Definition 5.13.** For an $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$-expression ($\mathsf{AL}^{\tau}_{\mathsf{cc}}$-expression, resp.) s, a position p, and a subexpression s' such that $s|_p = s'$ the *non-R-depth* of s' at p is the number of prefixes p' of p s.t. $s[\cdot]_{p'}$ is not a reduction context.

▶ **Lemma 5.14.** *For $D \in \{\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}, \mathsf{AL}^{\tau}_{\mathsf{cc}}\}$, a D-expression t, a D-context C with $C[t]{\downarrow_D}$ there is some i and an approximation $approx(t,i)$ with $C[approx(t,i)]{\downarrow_D}$.*

Proof. Let $C[t] \xrightarrow{n}_D t_0$, where t_0 is a D-WHNF. Then compute $t' := approx(t, n+1)$. The construction of $approx(t, n+1)$ includes (β)-, (case)-, (seq)- and (fix)-reductions and simplification rules. Let A be the set of all the simplification rules. We have $C[t] \xrightarrow{bcsfC \cup A, *} C[t'']$, where t' is t'' with subexpressions replaced by `Bot`. Since reductions and simplifications are correct, we have $C[t'']{\downarrow_D}$, and in particular, the number of normal order reductions of $C[t'']$ to a D-WHNF is $n' \leq n$ (proven in [15]).

The normal order reduction for $C[t']$ makes the same reduction steps as the normal order reduction of $C[t'']$ since the Bot-expressions placed by the approximation are in the beginning at the non-R-depth $n+1$, and remain at non-R-depth $\geq n+1-j$ after j normal order reductions. Finally, they will be at non-R-depth of at least 1, hence the final D-WHNF may have Bots only at non-R-depth of at least 1, and so it is a WHNF. Thus $C[approx(t,n)]{\downarrow_D}$. ◀

▶ **Theorem 5.15.** *The embeddings of* AL^τ *in* $\mathsf{AL}^\tau_{\mathsf{cc}}$ *and of* $\mathsf{AL}^\tau_{\mathsf{seq}}$ *in* $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$ *are conservative.*

Proof. We prove this for the embedding of $\mathsf{AL}^\tau_{\mathsf{seq}}$ in $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$. The other case is similar. Let s,t be $\mathsf{AL}^\tau_{\mathsf{seq}}$-expressions with $s \leq_{\mathsf{AL}^\tau_{\mathsf{seq}}} t$. We have to show that $s \leq_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}} t$. Assume this is false. Since the AP_i-context lemma holds (Theorem 5.1) the assumption implies that there is an $n \geq 0$ and closed $\mathsf{AL}^\tau_{\mathsf{cc,seq}}$-expressions $b_1,\ldots,b_n$ of $\mathsf{AL}^\tau_{\mathsf{seq}}$-type which are answers or Bot, such that $(s\ b_1 \ldots b_n){\downarrow_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}}}$ but $(t\ b_1 \ldots b_n){\Uparrow_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}}}$. According to Lemma 5.14, we have successively constructed the approximations b'_i of b_i of a depth depending on the length of the normal order reduction of $(s\ b_1 \ldots b_n)$, such that $(s\ b'_1 \ldots b'_n){\downarrow_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}}}$ but $(t\ b'_1 \ldots b'_n){\Uparrow_{\mathsf{AL}^\tau_{\mathsf{cc,seq}}}}$, also using Lemma 5.11. Lemma 5.12 shows that the approximations are in the smaller calculus $\mathsf{AL}^\tau_{\mathsf{seq}}$, and thus also $(s\ b'_1 \ldots b'_n){\downarrow_{\mathsf{AL}^\tau_{\mathsf{seq}}}}$ but $(t\ b'_1 \ldots b'_n){\Uparrow_{\mathsf{AL}^\tau_{\mathsf{seq}}}}$, which contradicts $s \leq_{\mathsf{AL}^\tau_{\mathsf{seq}}} t$. ◀

The same reasoning can be used to show the following result (of practical interest) for $D \in \{\mathsf{AL}^\tau_{\mathsf{cc}}, \mathsf{AL}^\tau_{\mathsf{cc,seq}}\}$: Assume that the set of type and data constructors is a fixed set in D, and that D' is an extension of D such that only new type and data constructors are added. Then D' is a conservative extension of D, since we can use the approximation technique from this section to approximate D'-values by D-values and then apply the AP_i-context lemma.

6 Polymorphically Typed Calculi

We consider polymorphically typed variants $\mathsf{AL}^\alpha, \mathsf{AL}^\alpha_{\mathsf{seq}}, \mathsf{AL}^\alpha_{\mathsf{cc}}, \mathsf{AL}^\alpha_{\mathsf{cc,seq}}$ of the four calculi. We will show non-conservativity of embedding AL^α in $\mathsf{AL}^\alpha_{\mathsf{seq}}$ and $\mathsf{AL}^\alpha_{\mathsf{cc}}$ in $\mathsf{AL}^\alpha_{\mathsf{cc,seq}}$, but leave open the question of (non-)conservativity of embedding AL^α in $\mathsf{AL}^\alpha_{\mathsf{cc}}$ and $\mathsf{AL}^\alpha_{\mathsf{seq}}$ in $\mathsf{AL}^\alpha_{\mathsf{cc,seq}}$.

The expression syntax is the untyped one. The syntax for polymorphic types $\overline{T}$ is $\overline{T} ::= V \mid \overline{T}_1 \to \overline{T}_2 \mid (K\ \overline{T}_1 \ldots \overline{T}_{ar(K)})$ where V is a type variable. The constructors have predefined Hindley-Milner polymorphic types according to the usual standards. Only expressions that are Hindley-Milner polymorphically typed are permitted. Normal order reduction is defined only on monomorphic type-instances of expressions, which is a deviation from Definition 2.1.

▶ **Definition 6.1.** For $D \in \{\mathsf{AL}^\alpha, \mathsf{AL}^\alpha_{\mathsf{seq}}, \mathsf{AL}^\alpha_{\mathsf{cc}}, \mathsf{AL}^\alpha_{\mathsf{cc,seq}}\}$ and for $s,t \in D$ of equal polymorphic type: $s \leq_D t$ iff $\rho(s) \leq_{D'} \rho(t)$ for all monomorphic type instantiations ρ, where D' is the corresponding monomorphically typed calculus. Contextual equivalence is defined by $s \sim_D t$ iff $s \leq_D t \wedge t \leq_D s$.

Since s_{11}, s_{12} (Sect. 5.1) are of polymorphic type $(a \to \mathtt{Bool}) \to \mathtt{Bool}$, the same arguments as for the proof of Theorem 5.6 can be applied, hence:

▶ **Theorem 6.2.** *The natural embedding of* $\mathsf{AL}^\alpha_{\mathsf{cc}}$ *into* $\mathsf{AL}^\alpha_{\mathsf{cc,seq}}$ *is not conservative.*

Let s_{13}, s_{14} of the polymorphic type $((\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha)) \to (\alpha \to \alpha))$ be defined as: $s_{13} := \lambda x.x\ id\ (x\ \mathtt{Bot}\ id)$ and $s_{14} := \lambda x.x\ id\ (x\ (\lambda y.\mathtt{Bot})\ id)$.

▶ **Lemma 6.3.** *For* AL^τ*-expressions* $t = M[\mathtt{Bot},\ldots,\mathtt{Bot}]$, $t' = M[\lambda x.\mathtt{Bot},\ldots,\lambda x.\mathtt{Bot}]$, *and* ${t\Uparrow_{\mathsf{AL}^\tau}}$, ${t'\downarrow_{\mathsf{AL}^\tau}}$ *it holds that* $M[x_1,\ldots,x_n] \xrightarrow{*}_{\mathsf{AL}^\tau} x_i$ *for some* i.

Proof. This follows by observing a normal order reduction of t, t' and comparing the first use of `Bot`, or $\lambda x.\texttt{Bot}$, respectively. There must be a use of this argument, since otherwise the observations are identical. If it is ever used in a function position in a beta-reduction, then both expressions diverge. Hence, the only possibility is that they are returned. ◀

▶ **Theorem 6.4.** *The embedding of* AL^{α} *into* $\mathsf{AL}^{\alpha}_{\mathsf{seq}}$ *is not conservative. The embedding of* AL^{α} *into* $\mathsf{AL}^{\alpha}_{\mathsf{cc,seq}}$ *is also not conservative.*

Proof. Since $(\rho(s_{13})\ (\lambda u, v.\texttt{seq}\ u\ v))\Uparrow_{\mathsf{AL}^{\tau}}$, but $(\rho(s_{14})\ (\lambda u, v.\texttt{seq}\ u\ v))\downarrow_{\mathsf{AL}^{\tau}}$ for $\rho = \{\alpha \mapsto o\}$, we have $s_{13} \not\sim_{\mathsf{AL}^{\alpha}_{\mathsf{seq}}} s_{14}$ as well as $s_{13} \not\sim_{\mathsf{AL}^{\alpha}_{\mathsf{cc,seq}}} s_{14}$. It remains to show that $s_{13} \sim_{\mathsf{AL}^{\alpha}} s_{14}$ holds, i.e. that $\rho(s_{13}) \sim_{\mathsf{AL}^{\tau}} \rho(s_{14})$ for any monomorphic type instantiation ρ of the type $((a \to a) \to (a \to a) \to (a \to a)) \to (a \to a))$. We use the AP_i-context lemma (Theorem 5.1) and assume that there is an n, a closed AL^{τ}-expression s, and closed arguments $b_1, \ldots, b_n$, such that $\rho(s_{13})\ s\ b_1 \ldots b_n$ is typed in AL^{τ}, and $\rho(s_{13})\ s\ b_1 \ldots b_n \Uparrow_{\mathsf{AL}^{\tau}}$, $\rho(s_{14})\ s\ b_1 \ldots b_n \downarrow_{\mathsf{AL}^{\tau}}$. By Lemma 6.3, the only possibility is that the `Bot`, and $\lambda x.\texttt{Bot}$-positions are extracted. By the type preservation, and since the type of $\rho(s_{13})\ s$ is the type of the `Bot`-position, it is impossible that $n > 0$, since then the type of the result is smaller than the type of the `Bot`-position. Hence $s\ id\ (s\ y\ id) \xrightarrow{*}_{\mathsf{AL}^{\tau}} y$. But since the y occurs in the expression $(s\ y\ id)$, we also have $(s\ y\ id) \xrightarrow{*}_{\mathsf{AL}^{\tau}} y$. This implies that $(s\ id\ y) \xrightarrow{*}_{\mathsf{AL}^{\tau}} y$. But then the normal order reduction of $s\ x_1\ x_2$ cannot apply either of its arguments x_1, x_2, and hence must be a projection to one of the arguments, which is impossible, since it must project to both arguments. We conclude that $\rho(s_{13})$ and $\rho(s_{14})$ cannot be distinguished in all approximation contexts, and the reasoning does not depend on ρ. Hence $s_{13} \sim_{\mathsf{AL}^{\alpha}} s_{14}$. ◀

The expressions s_{13}, s_{14} could also be used to show non-conservativity of embedding AL^{τ} into $\mathsf{AL}^{\tau}_{\mathsf{seq}}$. Hence there are also examples at higher types as witnesses for Theorem 5.6.

Whether adding case and constructors is conservative or not in the polymorphic case, for AL^{α} as well as for $\mathsf{AL}^{\alpha}_{\mathsf{seq}}$ remains an open problem.

Forgetting Types. Now we look for the translations defined as "forgetting" the types, and ask for adequacy and full abstraction, which plays now the role of conservativity. For the monomorphically typed calculi the answer is obvious: these translations are not fully abstract. For example $\lambda x^{o}.x^{o}$ is equivalent to $\lambda x^{o}.\bot^{o}$, which refutes full abstractness in all cases. For the polymorphically typed calculi, this question is non-trivial:

▶ **Proposition 6.5.** *The translations of* AL^{α} *into* AL, $\mathsf{AL}^{\alpha}_{\mathsf{cc}}$ *into* $\mathsf{AL}_{\mathsf{cc}}$, *and* $\mathsf{AL}^{\alpha}_{\mathsf{cc,seq}}$ *into* $\mathsf{AL}_{\mathsf{cc,seq}}$ *by simply forgetting the types are adequate but not fully-abstract.*

Proof. For the first case, AL^{α} and AL, we have $s_{13} \sim_{\mathsf{AL}^{\alpha}} s_{14}$, but $(s_{13}\ \lambda u, v.(v\ (\lambda x.u)))\Uparrow$ and $(s_{14}\ \lambda u, v.(v\ (\lambda x.u)))\downarrow$. For the other calculi, $\lambda x.\texttt{case}_{Bool}\ x\ (\texttt{True -> True})\ (\texttt{False -> False})$ is equivalent to $\lambda x^{\texttt{Bool}}.x$, but in the untyped case, $([\cdot]\ \lambda z.z)$ distinguishes these expressions. ◀

Full abstractness of forgetting types in $\mathsf{AL}^{\alpha}_{\mathsf{seq}}$ also remains an open question.

7 Conclusion

We have shown that the semantics of the pure lazy lambda calculus changes when `seq`, or `case` and constructors, are added. Under the insight that any semantic investigation for Haskell should include the `seq`-operator, we exhibited calculus extensions that are useful for the analysis of expression equivalences that also hold in a realistic core calculus of lazy functional and typed languages. We left the rigorous analysis of the implication chain for equivalence from $\mathsf{AL}^{\tau}_{\mathsf{cc,seq}}$ to the polymorphic calculus with letrec for future research.

References

1 S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116, 1990.
2 A. Arbiser, A. Miquel, and A. Rios. A lambda-calculus with constructors. In *Proc. RTA 2006*, volume 4098 of *LNCS*, pages 181–196, 2006.
3 R. C. de Vrijer. Extending the lambda calculus with surjective pairing is conservative. In *Proc. LICS 1989*, pages 204–215, 1989.
4 M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Programming*, 17(1–3):35–75, 1991.
5 S. Holdermans and J. Hage. Making "stricterness" more relevant. In *Proc. PEPM 2010*, pages 121–130, 2010.
6 D. Howe. Equality in lazy computation systems. In *Proc. LICS 1989*, pages 198–203, 1989.
7 D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.
8 P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fund. Inform.*, 69(1–2):63–102, 2006.
9 J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8:275–317, 1998.
10 A. K. D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL 1999*, pages 43–56, 1999.
11 S. Peyton Jones. *Haskell 98 language and libraries: the revised report.* 2003.
12 S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Programming*, 32(1–3):3–47, 1998.
13 J. G. Riecke and R. Subrahmanyam. Extensions to type systems can preserve operational equivalences. In *Proc. TACS 1994*, pages 76–95, 1994.
14 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *Proc. PPDP 2011*, pages 101–112, 2011.
15 M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending abramsky's lazy lambda calculus: (non)-conservativity of embeddings. Frank report 51, Inst. f. Informatik, Goethe-University, Frankfurt, 2013. available at http://www.ki.informatik.uni-frankfurt.de/papers/frank.
16 M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, volume 273, pages 521–535, 2008.
17 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *Proc. RTA 2010*, volume 6 of *LIPIcs*, pages 295–310, 2010.
18 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. Frank report 49, Inst. f. Informatik, Goethe-University, Frankfurt, 2012.
19 M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.
20 P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Programming*, 7(3):231–264, 1997.
21 K. Støvring. Extending the extensional lambda calculus with surjective pairing is conservative. *Log. Methods Comput. Sci.*, 2(2), 2006.
22 TLCA. List of open problems, 2010. `http://tlca.di.unito.it/opltlca/`.
23 P. Wadler. Theorems for free! In *Proc. FPCA 1989*, pages 347–359, 1989.

Algorithms for Extended Alpha-Equivalence and Complexity*

Manfred Schmidt-Schauß, Conrad Rau, and David Sabel

Goethe-Universität, Frankfurt, Germany
{schauss,rau,sabel}@ki.informatik.uni-frankfurt.de

Abstract

Equality of expressions in lambda-calculi, higher-order programming languages, higher-order programming calculi and process calculi is defined as alpha-equivalence. Permutability of bindings in let-constructs and structural congruence axioms extend alpha-equivalence. We analyse these extended alpha-equivalences and show that there are calculi with polynomial time algorithms, that a multiple-binding "let" may make alpha-equivalence as hard as finding graph-isomorphisms, and that the replication operator in the pi-calculus may lead to an EXPSPACE-hard alpha-equivalence problem.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases alpha-equivalence, higher-order calculi, deduction, pi-calculus

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.255

1 Introduction

Motivation. Reasoning, rewriting, matching, and automated deduction in higher order calculi often require – as a very basic operation – to identify higher-order expressions *up to alpha-equivalence*. This means expressions are identified if they are syntactically equal up to a renaming of bound variables (which represent the binding structure). As a basic example consider the expressions of the classical lambda calculus $e_1 = \lambda x.\lambda y.x$ and $e_2 = \lambda y.\lambda x.y$. These expressions are alpha-equivalent, since the renaming $\sigma = \{y \mapsto x, x \mapsto y\}$ of *bound* variables makes $\sigma(e_1)$ and e_2 syntactically equally. An approach to handle alpha-equivalence in deduction systems is to use nominal techniques [27], however, the focus is to ease formula specification and deduction rather than speeding up alpha-equivalence checking. In addition nominal techniques consider so-called equivariance between terms, which is a slight extension of alpha-equivalence, since terms e_1, e_2 are equivariant if there exists a finite permutation of variable names π such that e_1 is syntactically equal to πe_2.

For a lot of classical program calculi (e.g. several variants of extensions of the lambda calculus) checking alpha-equivalence can be performed by efficient (and also more or less trivial) algorithms in log-linear time in the size of the expressions. Also deciding equivariance of such terms is known to be in **P** [9].

However, more sophisticated calculi also allow programming primitives that satisfy laws like commutativity and / or associativity in combination with binding primitives, like non-recursive and recursive bindings, which may occur nested (e.g. `let(rec)`-expressions in extended lambda-calculi like [2, 23, 38], the parallel composition and ν-binders in process calculi, like the π-calculus [21, 20, 35]). Equality testing of expressions in such calculi,

* This work was supported by the DFG under grant SCHM 986/9-1 and SCHM 986/9-2

respecting alpha-equivalence and the laws of the primitives, turns out to have a harder decision problem.

In this paper we focus on this problem and describe several algorithms and also determine the complexity of checking *equality of expressions* up to alpha-equivalence under the usual laws of the programming primitives (permutativity, commutativity, associativity, etc.). Algorithms for expressions in higher-order extended lambda-calculi and process-calculi will be discussed, which also includes alpha-equivalence of functional programs in Haskell.

Applications. Our motivation to analyze extended alpha-equivalence stems from recent research that aims to automate the diagram-based proof method for showing correctness of program transformations (see [38] for the method and [29, 28] for the automation). The method is mainly used for call-by-need programming calculi modelling the semantics of functional programming languages like Haskell, however, the method is also applicable to other kinds of calculi like variations of call-by-need calculi, and to process calculi like the π-calculus. The first step is to compute critical overlaps (similar to critical pairs) between reduction rules from the operational semantics and program transformations using a sophisticated unification algorithm [29, 30]. In a second step the overlappings must be "closed", which is similar to show joinability of critical pairs. This requires to find out whether two reduction sequences starting from different expressions lead to expressions that are alpha-equivalent after permutation of bindings. Thus checking expressions for extended alpha-equivalence is an operation that is often performed even on large expressions. Ad-hoc algorithms for checking alpha-equivalence of such expressions are worst-case exponential due to searching for all possible permutations. Indeed, we will show in this paper that this is unavoidable in general. Another potential application of interest in program analysis during compilation of (functional) programs is in "common subexpression elimination" which shares identical subexpressions and where subexpressions must be checked for equality up to alpha-equivalence.

Graph isomorphism. Several extended alpha-equivalence problems will be shown to be graph-isomorphism-complete (**GI**-complete) in this paper. The graph-isomorphism problem as a complexity class **GI** is only known to be between **PTIME** and **NP**. Proving an algorithmic problem as **GI**-complete indicates hardness and (assuming current knowledge) that there is no polynomial time algorithm for it solving all instances. More details from a complexity point of view on the class **GI** can e.g. be found in [39, 16, 15], where it is also show that if **GI** were **NP**-complete, then the polynomial time hierarchy would collapse.

A related problem is term equality including associative-commutative operators which is shown to be **GI**-complete in [6]. However, [6] does not consider the case of (perhaps mutually-recursive) binding-environments as they are provided by the `letrec`-construct. Another result is that deciding structural congruence in the π-calculus *without replication* (but perhaps with recursion) is **GI**-complete [14]. Algorithms for deciding structural congruence in the π-calculus *with replication* were investigated for several variants of the calculus in [10, 11], where the complexity is either not analyzed or shown to be in **EXPSPACE**. As we show – under mild restrictions – the problem is also **EXPSPACE**-hard. A further related result is that syntactic equivalence of Boolean formulas in CNF using associativity/commutativity of Boolean operators is **GI**-complete [5].

Results and structure of the paper. Section 2 contains preliminaries on graphs, the isomorphism problem of various variants of graphs, and a proof of **GI**-completeness for

a special class of graphs (so-called outgoing-ordered labelled directed graphs). Moreover, an efficient decision procedure is presented for a subclass of these graphs. Based on these results, we prove in Section 3 that alpha-equivalence for a class of higher-order languages with letrec-expressions is **GI**-complete (Theorem 3.8), and that there is a polynomial time algorithm provided the expressions are free of garbage (Theorem 3.11) or rewrite rules for performing garbage collection are included in the calculus (Theorem 3.13). We also show **GI**-completeness for languages with non-recursive or non-nested let. We also present several instances of program calculi that are covered by our results. In Section 4 we investigate structural congruence in process calculi. Especially, we summarize known results about the complexity of structural congruence in several variations of the π-calculus and provide a proof of **EXPSPACE**-hardness of structural congruence of Milner's variant (Theorem 4.3).

2 Graphs and Graph Isomorphism

Before considering the (extended) alpha-equivalence problems, as a preliminary we introduce the necessary notions and notation on graphs and the graph isomorphism problem. In this section, we also introduce some specific, restricted graphs and their isomorphism problems together with algorithms and analyses of them. In later sections, these results will be applied to the alpha-equivalence problem in different program calculi.

We define labelled directed graphs as a flexible formalism for several classes of graphs, e.g. including unlabelled, undirected graphs.

▶ **Definition 2.1.** A *labelled directed graph (LDG)* is a tuple $G = (V, E, L, \text{lab})$ where V is a finite set of nodes, $E \subseteq (V \times V \times L)$ are directed labelled edges between nodes, L is a finite set of labels, V, E are disjoint, and $\text{lab} : V \to L$ assigns a label to every node. If $(v_1, v_2, l) \in E \iff (v_2, v_1, l) \in E$, then the graph is *undirected*, and if $|L| = 1$, then we call G *unlabelled*. For convenience, we omit the components L and lab for unlabelled graphs.

Note that this definition does only allow parallel edges with different labels, and forbids parallel edges in unlabelled graphs. An isomorphism between two LDGs is defined as follows:

▶ **Definition 2.2** (Isomorphic LDGs)**.** Two LDGs $G_1 = (V_1, E_1, L, \text{lab}_1)$, $G_2 = (V_2, E_2, L, \text{lab}_2)$ are *isomorphic* iff there is a bijection $\phi : V_1 \to V_2$ such that $(v_1, v_2, l) \in E_1 \iff (\phi(v_1), \phi(v_2), l) \in E_2$ for all edges $(v_1, v_2, l) \in E_1$ and $\text{lab}_1(v) = \text{lab}_2(\phi(v))$ for all nodes $v \in V_1$. The mapping ϕ is called an *isomorphism* in this case.

Note that an isomorphism is completely determined by the mapping on the nodes.

▶ **Definition 2.3** (Graph Isomorphism Problem (GI))**.** Graph-isomorphism (**GI**) is the following problem: Given two finite (unlabelled, undirected) graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, are G_1 and G_2 isomorphic?

It is well-known that the isomorphism problem for *directed* graphs is **GI**-complete [41], which we will use later for our encodings. Also, the labelled directed graph isomorphism problem is equivalent to the unlabelled graph isomorphism problem, i.e. it is known to be **GI**-complete. See [7] for further classes of graphs with a **GI**-complete isomorphism problem. We summarize these known results in the following proposition:

▶ **Proposition 2.4** ([7, 41])**.** *The isomorphism problem for LDGs and the isomorphism problem for unlabelled directed graphs are* **GI***-complete.*

Figure 1 Example for the encoding ρ in the proof of Proposition 2.6 (new nodes are shaded).

The graph-isomorphism problem as a complexity class **GI** is only known to be between **PTIME** and **NP**.

In our application in the next section, the following specific graphs play an important role. The graphs are LDGs where all outgoing edges of a node have unique labels:

▶ **Definition 2.5** (Outgoing-Ordered LDG)**.** We call a labelled directed graph $G = (V, E, L, \text{lab})$ *outgoing-ordered* (an *OOLDG*), iff for every node $v \in V$: whenever $(v, v_1, l), (v, v_2, l) \in E$ then $v_1 = v_2$.

OOLDGs are related to expressions with a restricted use of `let`-expressions, where the intuition is that their alpha-equivalence-check may be more efficient.

The paper [12] describes algorithms for matching directed labelled graphs, but the isomorphism definition is w.r.t. the (infinite) unrolling, and thus different. The isomorphism problem for so-called ordered directed graphs was shown to be solvable in polynomial time by [13]. However, this result is only applicable to labelled directed graphs where for every node the – outgoing as well as the incoming – edges are uniquely labelled for every node. For our application, the incoming edges have no restrictions. Hence their result cannot be used. The following proposition even shows that these questions are different:

▶ **Proposition 2.6.** *The isomorphism problem for OOLDGs is* **GI***-complete.*

Proof. The problem is in **GI** (this trivially follows from Proposition 2.4). For proving **GI**-hardness, we define an encoding ρ of arbitrary unlabelled directed graphs into OOLDGs: Given an unlabelled directed graph G, we construct an OOLDG $\rho(G)$ from G by the following operation: Every edge (v_1, v_2) of G is replaced by two edges $(v, v_1, 1)$ and $(v, v_2, 2)$, where v is a new node: i.e. $v_1 \to v_2$ is replaced by $v_1 \xleftarrow{1} v \xrightarrow{2} v_2$, where for every edge a fresh node v is constructed. An example of the encoding is shown in Figure 1.

Given two unlabelled directed graphs G_1, G_2, these are isomorphic iff $\rho(G_1)$ and $\rho(G_2)$ are isomorphic: Any isomorphism of $\rho(G_1)$ and $\rho(G_2)$ only maps old nodes to old ones and fresh nodes to fresh ones due to the direction of the new edges (i.e. only new nodes have outgoing edges, and only old nodes have incoming edges). The direction of the edges is preserved (i.e. encoded) due to the labels $1, 2$. Since the encoding ρ at most doubles the size of the graphs and the isomorphism problem for (unlabelled) directed graphs is **GI**-complete (Proposition 2.4), OOLDG-isomorphism checking is **GI**-hard. ◀

Since the encoding generates an *acyclic* directed graph, this also implies:

▶ **Proposition 2.7.** *The isomorphism problem for acyclic OOLDGs is* **GI***-complete.*

However, as we show in the remainder of this section, there are also some complexity results for OOLDGs, which are positive under various reachability restrictions. First, we define some notation:

▶ **Definition 2.8.** Let $G = (V, E, L, \text{lab})$ be a (labelled or unlabelled) directed graph. We say a node $v \in V$ is a *root*, if it has no incoming edges (there exists no edge $(w, v) \in E$), and it is a *leaf* if it has no outgoing edges (there exists no edge $(v, w) \in E$).

For a node v of G, let $\text{reach}(v, G)$ be the set of nodes in G reachable from v via directed edges. A node v is called *initial*, iff every other node $w \in V \setminus \{v\}$ can be reached from v (i.e. $w \in \text{reach}(v, G)$). A set S of nodes is *initial*, iff every other node $w \in V \setminus S$ can be reached from some $v \in S$ (i.e. $w \in \bigcup_{v \in S} \text{reach}(v, G)$).

G is *weakly connected*, iff its corresponding undirected graph is connected. Hence G can be split into its weakly connected components. We say G is *k-initial*, if for every weakly connected component G' of G, there is an initial set $S_{G'}$ within G' of at most k nodes (i.e. $|S_{G'}| \leq k$). We say G is *k-rooted*, if every weakly connected component G' of G has at most k roots, and if the set of roots within G' is initial. We say G is *rooted*, if it is 1-rooted, and G is weakly connected. For an LDG $G = (V, E, L, \text{lab})$ its *size* $|G|$ is the sum of the cardinalities of V and E.

▶ **Proposition 2.9.** *Let G_1, G_2 be rooted OOLDGs. Then isomorphism between G_1 and G_2 can be tested in time $O(n \log(n))$ where $n = |G_1| + |G_2|$. Moreover, there is at most one isomorphism between G_1 and G_2.*

Proof. Let $G_i = (V_i, E_i, L_i, \text{lab}_i)$ for $i = 1, 2$. W.l.o.g, we can assume that for $i = 1, 2$: every $l \in L_i$ is used in G_i. If $L_1 \neq L_2$, then G_1 and G_2 are not isomorphic, hence we assume $L_1 = L_2$. We construct a mapping $\phi : V_1 \to V_2$ such that either ϕ is an isomorphism between G_1 and G_2 or the construction fails. In the latter case G_1 and G_2 are not isomorphic. Let r_i be the root nodes of G_i, for $i = 1, 2$. We set $\phi(r_1) := r_2$. We iteratively process all nodes in V_1: Assume $\phi(v_1) = w_1$ is already constructed, and the outgoing edges are $(v_1, v_{1,j}, l_j)$ for $j = 1, \ldots, m$. Then the outgoing edges from w_1 either can be ordered as $(w_1, w_{1,j}, l_j)$ for $j = 1, \ldots, m$, or the ϕ-construction fails and G_1, G_2 are not isomorphic. We set $\phi(v_{1,j}) := w_{1,j}$ for $j = 1, \ldots, m$. If there is a conflict, for example since ϕ is already differently defined on some $v_{1,k}$, then again the construction fails, and G_1, G_2 are not isomorphic. If the construction goes through without failing, then an isomorphism has been found, since the graphs are rooted, and hence every node is reachable. Obviously, the construction leads to a unique ϕ, if the construction halts successfully. As a preprocessing we store all edges (w_1, w_2, l_j) of E_2 in an efficient data structure with key (w_1, l_j) and value w_2. This can be done in $O(|E_2| \log |E_2|)$ time. During the construction of ϕ we lookup the corresponding edge in $O(\log |E_2|)$ time. The mapping ϕ can also be stored in an efficient data structure, which shows that the whole procedure requires $O(n \log n)$ time. ◀

▶ **Proposition 2.10.** *Let G_1, G_2 be OOLDGs with $n = |G_1| + |G_2|$. Estimations for the complexity of isomorphism checking of G_1, G_2 are:*

1. *$O(k!\ n \log(n))$, if G_1, G_2 are weakly connected and k-rooted.*
2. *$O(k!\ n^3 \log(n))$, if G_1, G_2 are k-rooted.*
3. *$O(k!\ n^2 \log(n))$, if G_1, G_2 are weakly connected and k-initial.*
4. *$O(k!\ n^4 \log(n))$, if G_1, G_2 are k-initial.*

Proof. The first item for $k = 1$ is exactly Proposition 2.9. For $k > 1$, all possible bijections of the roots have to be tried, which justifies the factor $k!$. The complexity in the second item is derived from the previous item, since $O(n^2)$ isomorphism checks between the weakly-connected components have to be performed followed by a comparison of the equivalence classes. Since the initial nodes are not unique in contrast to the roots, also all possibilities for other nodes have to be tried, which increases the exponent of n by 1 in the corresponding cases (3) and (4). ◀

▶ **Definition 2.11.** For an LDG $G = (V, E, L, \text{lab})$, we define the *outgoing-ordered subgraph (OO-subgraph)* $OO(G) = (V, E', L, \text{lab})$, which is constructed from G by removing all the edges with ambiguous labels. More rigorously: for every node $v \in V$, and every label $l \in L$, if $e_1, \ldots, e_n$ are all the outgoing edges from v, labelled with l, and $n \geq 2$, then remove $e_1, \ldots, e_n$, but not the nodes. The resulting graph is denoted as $OO(G)$.

▶ **Proposition 2.12.** *Let G_1, G_2 be two LDGs such that $G_i' := OO(G_i)$ for $i = 1, 2$ are rooted. Then isomorphism of G_1, G_2 can be tested in time $O(n \log(n))$ where $n = |G_1| + |G_2|$. Moreover, there is at most one isomorphism.*

Proof. Any isomorphism ϕ between G_1 and G_2 is also an isomorphism between G_1' and G_2', since $OO(\cdot)$ does not remove nodes. Moreover, any isomorphism ϕ between G_1' and G_2' is a bijective mapping between the nodes of G_1 and G_2. Using Proposition 2.9, the isomorphism test of G_1', G_2' can be performed in time $O(n \log(n))$, where the mapping ϕ is also constructed on the fly. Then we test whether ϕ is an isomorphism of G_1, G_2 which can also be performed in time $O(n \log n)$. ◀

3 Alpha-Equivalence for Higher-Order Languages with Let

We define a fragment of the core language of higher order extended lambda-calculi with a recursive or non-recursive let that captures the essence at least of its alpha-equivalence issues and is nevertheless general enough such that the results grade up to the full language.

A *signature* Σ is a finite set of ranked constructor or function symbols c, c_i equipped with an arity $ar(c) \in \mathbb{N}_0$. We assume a countable infinite set of variables denoted by x, y, z (possibly indexed by numbers). The language CH (over the signature Σ) has the syntax

$$r, s, t \in \mathcal{L}_{\mathsf{CH}} ::= x \mid c(s_1, \ldots, s_{ar(c)}) \mid \lambda x.s \mid \texttt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \texttt{in}\ s.$$

The constructs that bind variables are abstractions (λ-expressions) and `letrec`-expressions. In a letrec-construct $\texttt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \texttt{in}\ s$, the bindings $x_i = s_i$ may be interchanged; the variables $x_1, \ldots, x_n$ must be distinct; the scope of the bound variables x_i is in every $s_1, \ldots, s_n$ as well as in s. As an abbreviation we use *Env* (as a meta symbol) for a (non-empty) set of `letrec`-bindings, e.g. we sometimes write $\texttt{letrec}\ Env\ \texttt{in}\ s$, or also $\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ s$. As another abbreviation we use $\lambda x_1, \ldots, x_n.s$ instead of $\lambda x_1. \ldots . \lambda x_n.s$. For a CH-expression s we use $FV(s)$ for the set of *free variables* of s.

Several usual programming language constructs of extended lambda calculi like application, `seq`-expression, non-deterministic choice fit into this syntax by choosing a corresponding function symbol c for the construct. Also calculi with case-expressions are covered, where the usual case-expression $\texttt{case}\ s\ \texttt{of}\ (c_1(x_{1,1}, \ldots, x_{1,ar(c_1)}) \to t_1)\ \ldots\ (c_n(x_{1,1}, \ldots, x_{n,ar(c_n)}) \to t_n)$ is represented as $\texttt{case}(s, \lambda x_{1,1}, \ldots, x_{1,ar(c_1)}.t_1, \ldots, \lambda x_{n,1}, \ldots, x_{n,ar(c_n)}.t_n)$. *Alpha-equivalence* of CH-expressions is defined as follows and includes permutation of the letrec-bindings:

▶ **Definition 3.1.** Two CH-expressions s, t are *alpha-equivalent*, $s \simeq_{\alpha,\mathsf{CH}} t$ iff s can be transformed into t using perhaps several of the two operations: (i) renaming of bound variables without capture, (ii) permuting the bindings in the letrec-environments.

For example, $(\texttt{letrec}\ x = y; y = z\ \texttt{in}\ x) \simeq_{\alpha,\mathsf{CH}} (\texttt{letrec}\ x = z; y = x\ \texttt{in}\ y)$, where we have to exchange x, y by a substitution and then to permute the bindings in the environment.

The language CHNR is the same language as CH except that `letrec` is non-recursive. For convenience, in this case we write `let` instead of `letrec`. The scope of x_i in $\texttt{let}\ x_1 =$

$s_1; \ldots; x_n = s_n$ `in` t is only t, and in CHNR the alpha-equivalence is defined accordingly, where only the let-renamings are different due to the scoping.

It is known that alpha-equivalence of expressions in the lambda-calculus and its extensions without permutations of bindings is decidable in polynomial time, even in time $O(n \log n)$, where n is the size of the expressions [8].

The issue when checking alpha-equivalence of CH-expressions is the potential exponential time requirement in searching for all possible permutations of letrec-bindings. It is easy to see that the alpha-equivalence problem is in **NP**: A single guess for the permutation of every letrec-expression and then checking alpha-equivalence in polynomial time are sufficient.

3.1 Complexity in the General Case of CH and CHNR

First we show that alpha-equivalence in CH is **GI**-hard.

▶ **Proposition 3.2.** *Even if the signature Σ is empty, deciding* CH*-alpha-equivalence as well as* CHNR*-alpha-equivalence is* **GI***-hard.*

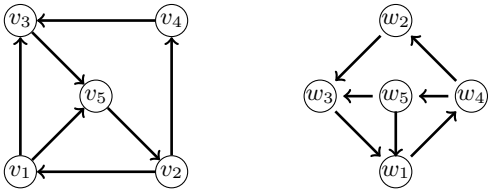
Proof. We encode the graph isomorphism problem for unlabelled directed graphs into the CHNR-alpha-equivalence problem. Hence, we encode two arbitrary unlabelled directed graphs $G_i = (V_i, E_i)$ for $i = 1, 2$ into CHNR-expressions $exp(G_i)$. We first assume that Σ contains a constant a and a binary constructor c, later we cover the case that Σ is empty. W.l.o.g. we assume $V_1 = \{v_{1,1}, \ldots, v_{1,n}\}$ and $V_2 = \{v_{2,1}, \ldots, v_{2,n}\}$ such that $V_1 \cap V_2 = \emptyset$.

The expression $exp(G_i)$ is `let` $Env_{i,A}$ `in` (`let` $Env_{i,B}$ `in` a), where the environments are as follows: For every node $v_{i,j}$ of V_i we have a component $v_{i,j} = a$ in $Env_{i,A}$, and for every edge $(v_{i,j}, v_{i,k})$ in E_i, we have the component $x_{i,j,k} = c(v_{i,j}, v_{i,k})$ in $Env_{i,B}$.

Assume that $exp(G_1) \simeq_{\alpha,\text{CHNR}} exp(G_2)$. Then there is a renaming $\sigma : V_1 \cup \bigcup\{x_{1,j,k}\} \to V_2 \cup \bigcup\{x_{2,j,k}\}$, such that $\sigma(exp(G_1))$ is syntactical equal to $exp(G_2)$ after some permutations of `let`-bindings. Let σ' be the restriction of σ to V_1. Obviously, σ' must be a bijection between V_1 and V_2; and $(v_{1,j}, v_{1,k}) \in E_1$ whenever $(\sigma'(v_{1,j}), \sigma'(v_{1,k})) \in E_2$. Thus σ' is an isomorphism between G_1 and G_2. Now assume that G_1 and G_2 are isomorphic. Then there exists a bijection $\sigma : V_1 \to V_2$ such that $(v_{1,j}, v_{1,k}) \in E_1$ iff $(\sigma(v_{1,j}), \sigma(v_{1,k})) \in E_2$. Then $exp(G_1)$ and $exp(G_2)$ are alpha-equivalent: σ can be used as a renaming, but has to be extended on the variables $x_{1,j,k}$ which is always possible. Thus G_1 and G_2 are isomorphic iff $exp(G_1) \simeq_{\alpha,\text{CHNR}} exp(G_2)$. Since every CHNR-expression is also a CH-expression, this also shows **GI**-hardness of CH-alpha-equivalence.

Now assume $\Sigma = \emptyset$: Then the same proof can be performed by replacing a by a free variable x_a and replacing $c(v_i, v_j)$ by `let` $y = v_i$ `in` v_j. ◀

▶ **Example 3.3.** We encode the following directed graphs as CHNR-expressions:



G_1 G_2

The encodings are $s_i = exp(G_i)$ with:

$$\begin{aligned} s_1 &= \texttt{let } v_1 = a; v_2 = a; v_3 = a; v_4 = a; v_5 = a \texttt{ in} \\ &\quad \texttt{let } x_1 = c(v_1, v_3); x_2 = c(v_1, v_5); x_3 = c(v_2, v_4); x_4 = c(v_2, v_1); \\ &\quad\quad x_5 = c(v_3, v_5); x_6 = c(v_4, v_3); x_7 = c(v_5, v_2) \texttt{ in } a \\ s_2 &= \texttt{let } w_1 = a; w_2 = a; w_3 = a; w_4 = a; w_5 = a \texttt{ in} \\ &\quad \texttt{let } x_1 = c(w_1, w_4); x_2 = c(w_2, w_3); x_3 = c(w_3, w_1); x_4 = c(w_4, w_2); \\ &\quad\quad x_5 = c(w_4, w_5); x_6 = c(w_5, w_1); x_7 = c(w_5, w_3) \texttt{ in } a \end{aligned}$$

The expressions are alpha-equivalent: For $\sigma = \{v_1 \mapsto w_5, v_5 \mapsto w_1, v_2 \mapsto w_4, v_4 \mapsto w_2, v_3 \mapsto w_3, x_1 \mapsto x_7, x_2 \mapsto x_6, x_5 \mapsto x_3, x_3 \mapsto x_4, x_4 \mapsto x_5, x_6 \mapsto x_2, x_7 \mapsto x_1\}$ the expression $\sigma(s_1)$ is syntactically equal to s_2 after "sorting" the `let`-environments.

The mapping σ restricted to $v_1, \ldots, v_5$ is an isomorphism between the graphs G_1, G_2.

▶ **Proposition 3.4.** *If Σ contains a binary constructor or function symbol, deciding alpha-equivalence in* CH *is* **GI***-hard, even if the expressions are restricted s.t.* `letrec` *is only allowed on the top-level of any expression, i.e. nested* `letrec`*-expressions are not permitted.*

Proof. The same proof as of Theorem 3.2 can be used, except that the expression encoding is `letrec` $Env_{i,A}, Env_{i,B}$ `in` a. ◀

Proposition 3.4 does not hold in CHNR, i.e., for non-recursive `let`-bindings. In this case alpha-equivalence can be decided in polynomial time which we show in Corollary 3.15.

Now we show that every alpha-equivalence problem in CH can be encoded as a directed graph-isomorphism problem, where the encoding can be done in polynomial time, and even in logarithmic space, i.e. deciding $\simeq_{\alpha,\mathsf{CH}}$ is in **GI**. First we define a graph construction from an expression:

▶ **Definition 3.5.** Given a CH-expression s, we describe the construction of $G(s)$, the labelled directed graph corresponding to s. We assume that s fulfills the distinct variable convention, i.e. the set of bound variables is distinct from free variables and bound variables are pairwise distinct. During the construction we use (and construct) a helper function `node`(.) which computes a node of the graph for every subexpression (with its position) of s. First the variables in s are partitioned into three sets: Let $\{x_1, \ldots, x_k\}$ be the free variables, $\{y_1, \ldots, y_m\}$ be the lambda-bound variables, $\{z_1, \ldots, z_n\}$ be the letrec-bound variables, and let $Var(s)$ be the union of the three sets.

Let $c_1, \ldots, c_k$ be the constructor and function symbols occurring in s, and q be the maximum arity of these symbols. Then the LDG $G(s)$ has the label set $L = \{\mathsf{var}, \mathsf{lamvar}, \mathsf{lvar}, \mathsf{body}, \mathsf{letvar}, \mathsf{bind}, \mathsf{letrec}, \mathsf{in}, \lambda\} \cup \{c_1, \ldots, c_k\} \cup \{1, \ldots, q\} \cup \{x_1, \ldots, x_k\}$.

For every variable $w \in Var(s)$ there is a node $N(w)$ in the graph $G(s)$. We set $\texttt{node}(w) := N(w)$. For every free variable x_i we set $\mathrm{lab}(N(x_i)) := x_i$, for every lambda-bound variable y_i we set $\mathrm{lab}(N(y_i)) = \mathsf{lamvar}$, and for every letrec-bound variable z_i we set $\mathrm{lab}(N(z_i)) = \mathsf{letvar}$.

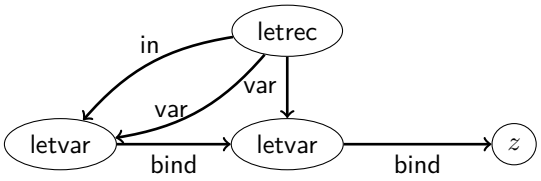
For the construction of the graph every subexpression is inspected (performed bottom up), according to the following cases (variables are already treated above):

- If the subexpression is $\lambda x.r$, then construct a new node v with $\mathrm{lab}(v) := \lambda$, and add edges $(v, \texttt{node}(x), \mathsf{lvar})$, $(v, \texttt{node}(r), \mathsf{body})$. Set $\texttt{node}(\lambda x.r) := v$.
- If the subexpression is ($\texttt{letrec } x_1 = s_1; \ldots; x_n = s_n \texttt{ in } t$), then we add one new node u with $\mathrm{lab}(u) := \mathsf{letrec}$, and let $\texttt{node}(\texttt{letrec } x_1 = s_1; \ldots; x_n = s_n \texttt{ in } t) := u$. For all $i = 1, \ldots, n$ we add edges $(u, \texttt{node}(x_i), \mathsf{var})$, $(\texttt{node}(x_i), \texttt{node}(s_i), \mathsf{bind})$, and an edge $(u, \texttt{node}(t), \mathsf{in})$.

- If the subexpression is $c(s_1, \ldots, s_n)$ then add one new node u with $\mathrm{lab}(u) = c$, and set $\mathtt{node}(c(s_1, \ldots, s_n)) := u$. For $i = 1, \ldots n$ add the edges (u, v_i, i).

Note that the construction of $\mathtt{node}(.)$ is meant together with the position of the subexpression, with the exception that for variables the position does not play any role, since there is only one node for every variable.

▶ **Example 3.6.** For the CH-expression $\mathtt{letrec}\ x = y, y = z\ \mathtt{in}\ x$ its corresponding LDG is $G = (V, E, L, \mathrm{lab})$ where $L = \{\mathsf{var}, \mathsf{lamvar}, \mathsf{lvar}, \mathsf{body}, \mathsf{letvar}, \mathsf{bind}, \mathsf{letrec}, \mathsf{in}, \lambda, z\}$, $V = \{v_1, v_2, v_3, v_4\}$, $E = \{(v_4, v_1, \mathsf{var}), (v_4, v_2, \mathsf{var}), (v_1, v_2, \mathsf{bind}), (v_2, v_3, \mathsf{bind}), (v_4, v_1, \mathsf{in})\}$, and $\mathrm{lab} = \{v_1 \mapsto \mathsf{letvar}, v_2 \mapsto \mathsf{letvar}, v_3 \mapsto z, v_4 \mapsto \mathsf{letrec}\}$. The graph can be depicted as follows:



▶ **Proposition 3.7.** CH-*alpha-equivalence is in* **GI**.

Proof. We encode alpha-equivalence of CH-expressions into the isomorphism problem of labelled directed graphs (which is GI-complete, see Proposition 2.4).

Let t_1, t_2 be CH-expressions. W.l.o.g. we assume that the expressions fulfill the distinct variable convention (if not, this can be achieved in time $O(n \log m)$ where n is the size of the term, and m is the number of variables). Let $G(t_1), G(t_2)$ be the graphs constructed according to Definition 3.5.

One can verify that t_1 and t_2 are alpha-equivalent if, and only if $G(t_1)$ and $G(t_2)$ are isomorphic: The whole term structure is preserved, `letrec`-bindings are commutable in the graph, and variable occurrences are represented by edges in the graph. Given an isomorphism ϕ from $G(t_1)$ to $G(t_2)$, the variable renaming making t_1 and t_2 syntactically equivalent (modulo commutation of `letrec`-bindings) can be derived by inspecting $\phi(\mathtt{node}(w))$ for every lambda- or letrec-bound variable w of t_1. This gives a node $v \in G(t_2)$ which must correspond to the according lambda- or letrec-bound variable in t_2. ◀

Obviously, CH-alpha-equivalence also solves CHNR-alpha-equivalence. Thus Propositions 3.7 and 3.2 imply:

▶ **Theorem 3.8.** CH-*alpha-equivalence and* CHNR-*alpha-equivalence are* **GI**-*complete.*

3.2 An Efficient Algorithm for Expressions without Garbage

The results up to now show that general CH-expressions have a hard alpha-equivalence problem. However, the question whether two CH-expressions are alpha-equivalent up to removing unused bindings can be answered efficiently, as we will see. Concretely, we define the following rewriting rules on CH-expressions for garbage collection (gc) that iteratively remove unused bindings:

$$(gc1)\quad \mathtt{letrec}\ x_1 = s_1; \ldots; x_n = s_n; y_1 = t_1; \ldots; y_m = t_m\ \mathtt{in}\ t_{m+1} \xrightarrow{gc} \mathtt{letrec}\ y_1 = t_1; \ldots; y_m = t_m\ \mathtt{in}\ t_{m+1} \quad \text{if } \bigcup_{i=1}^{m+1} FV(t_i) \cap \{x_1, \ldots, x_n\} = \emptyset$$

$$(gc2)\quad \mathtt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \mathtt{in}\ t \xrightarrow{gc} t \quad \text{if } FV(t) \cap \{x_1, \ldots, x_n\} = \emptyset$$

It is easy to verify that the rewriting system on CH-expressions induced by $\xrightarrow{gc}$ is confluent and terminating. Thus unique normal forms w.r.t. (gc) exist. We say a CH-expression s is *without garbage* (or *garbage free*), if it is such a normal form, i.e. the rule (gc) is not applicable to any subexpression of s. Computing the normal form w.r.t. (gc) can be done in polynomial time, since (gc)-redexes can be detected efficiently and the rewriting can be performed by an innermost-strategy, inspecting every `letrec`-expression once. The complexity of computing the (gc)-normal form using the (gc)-rewriting steps is polynomial in n where n is the size of the expressions. However, by a global procedure an (alpha-equivalent) (gc)-normal form can also be computed in time $O(n \log n)$:

▶ **Lemma 3.9.** *Let s be a* CH*-expression. Then an alpha-equivalent (gc)-normal form of s can be computed in time $O(n \log n)$ where n is the size of s.*

Proof. W.l.o.g. we assume that s fulfills the distinct variable convention (otherwise a renamed expression can be computed in time $O(n \log n)$). As a first step, construct the LDG $G(s)$ according to Definition 3.5. Then mark all nodes of $G(s)$ that are reachable from $\texttt{node}(s)$ by never using any edge labelled var. This requires time $O(|G(s)| \log |G(s)|)$ if we store the nodes in an efficient data structure. Clearly, unmarked nodes are garbage. Hence, we delete all unmarked nodes, all corresponding edges from $G(s)$, and finally letrec-nodes that have no outgoing edges marked with var, where incoming edges are redirected to the node corresponding to the `in`-expression. Let the result be the LDG G'. The deletion procedure requires $O(|G(s)| \log |G(s)|)$ time, since a traversal of the graph (with some lookups whether nodes are marked or not marked) is sufficient. Finally, reconstruct the garbage-free expression corresponding to G' which is always possible. ◀

Below we show that the alpha-equivalence-problem for garbage free CH-expressions can be decided efficiently, and thus also the question whether for two given expressions s_1, s_2 their (gc)-normal forms s_1', s_2' are alpha-equivalent can be answered in polynomial time.

Hence, the conclusion is that the worst-case high complexity for checking alpha-equivalence in CH is due to comparing the garbage subexpressions.

▶ **Lemma 3.10.** *Let s be a garbage free* CH*-expression, and $G(s)$ the LDG constructed from s (see Definition 3.5). Then the subgraph $G' := OO(G(s))$ satisfies the assumptions of Proposition 2.12: G' is rooted and an OOLDG.*

Proof. It is easy to see that garbage-freeness of s implies that every node of $G(s)$ is reachable from the root via edges that are not labelled with var. The subgraph $G' := OO(G(s))$ is the subgraph of $G(s)$ where for the nodes all var-edges are removed, if there are at least two outgoing var-edges from this node. ◀

▶ **Theorem 3.11.** *Given* CH*-expressions s_1, s_2, where at least one of s_1, s_2 is free of garbage, their alpha-equivalence can be checked in time $O(n \log(n))$ where $n = |G(s_1)| + |G(s_2)|$.*

Proof. It can be checked in log-linear time whether (gc) is applicable to an expression by constructing $G(s_i)$ and checking whether every node is reachable via edges not labelled with var. The expressions can only be alpha-equivalent, if both are free of garbage. Thus by Lemma 3.10 and Proposition 2.12 the isomorphism-test of $G(s_1)$ and $G(s_2)$ can be performed in time $O(n \log(n))$ where $n = |G(s_1)| + |G(s_2)|$. Since this is equivalent to alpha-equivalence of s_1, s_2, the theorem holds. ◀

▶ **Definition 3.12.** CH-expressions s, t are *alpha-equivalent up to garbage-collection*, denoted by $s \simeq_{\alpha,gc,\mathsf{CH}} t$, iff the (gc)-normal forms s' and t' of s and t are alpha-equivalent.

▶ **Theorem 3.13.** *For* CH*-expressions* s_1, s_2 *it is possible to decide whether* $s_1 \simeq_{\alpha,gc,\mathsf{CH}} s_2$ *in time* $O(n \log n)$ *where* $n = |s_1| + |s_2|$.

Proof. First, alpha-equivalent (gc)-normal forms of s_1, s_2 are computed by Lemma 3.9. Finally, Theorem 3.11 is applied to the (gc)-normal forms. ◀

Note that checking $\simeq_{\alpha,gc,\mathsf{CH}}$ instead of $\simeq_{\alpha,\mathsf{CH}}$ is an option for the automation of the diagram method, but requires the transformation (gc) in the set of permitted transformations/reductions in the reduction sequences, which is often not the case.
After adapting the (gc)-definition to CHNR, very similar reasoning and arguments show:

▶ **Theorem 3.14.** *Theorem 3.11 and 3.13 also hold for* CHNR.

▶ **Corollary 3.15.** *For* CHNR*-expressions* s_1, s_2 *where* `let`*-expressions are only allowed on the top-level of expressions, it is possible to decide whether* $s_1 \simeq_{\alpha,\mathsf{CHNR}} s_2$ *in time* $O(n^3 \log n)$ *where* $n = |s_1| + |s_2|$.

Proof. First decide whether $s_1 \simeq_{\alpha,gc,\mathsf{CHNR}} s_2$ using Theorem 3.14. Then compute the bindings of s_1 and s_2 that are garbage. Testing whether these bindings are alpha-equivalent is possible in time $O(n^3 \log n)$: alpha-equivalence of a single binding $x = t_1$ of s_1 and a single binding $y = t_2$ of s_2 can be tested in time $O(n \log n)$, since t_1, t_2 are usual terms (i.e. ranked trees). For testing the sets of bindings we have to compare $O(n^2)$ bindings. ◀

For CH-expressions s, t with garbage, but without nested letrecs, one can test first the expressions after removing the garbage bindings, and then test all combinations of the garbage-bindings. This implies the following complexity estimation:

▶ **Corollary 3.16.** *Let* s_1, s_2 *be* CH*-expressions with a one-level top letrec. Let* $n = |s_1| + |s_2|$ *and* k *be the number of let-variables that are garbage in* s_1*. Then* $s_1 \simeq_{\alpha,\mathsf{CH}} s_2$ *can be decided in time* $O(k!\, n \log n)$.

3.3 Applications: Lambda-Calculi with Bindings and Haskell

In this section we analyze and list several program calculi and programming languages which fit into the syntax of CH or CHNR and thus have a **GI**-complete alpha-equivalence problem.

The first class of calculi covers several call-by-need lambda calculi with letrec, all of them fit into the syntax of the language CH.

▶ **Proposition 3.17.** *Deciding alpha-equivalence in lambda-calculi with letrec is* **GI***-complete. This includes the following calculi: The call-by-need lambda calculi in [3, 2] (in the variants with letrec), the call-by-need lambda calculus with letrec in [37], and the cyclic lambda calculi [4, 1]. Extended call-by-need letrec calculi as e.g. used in [23, 38], and also extensions with non-deterministic operators [22, 24, 33, 36].*

In all the mentioned calculi a rule for garbage collection can be defined such that Theorem 3.13 is applicable.

Note that in call-by-need lambda calculi with non-recursive, single-binding let-expressions (as e.g. [18]) alpha-equivalence can be decided in log-linear time.

We analyze the following fragment of Haskell. Data definitions are permitted where the names of types and constructors are not renameable. Supercombinator definitions are also permitted, but no modules, expressions are built by the usual constructs: variables, abstractions, applications, constructor applications and case-expressions, also `letrec`-expressions may be present. We also assume that `main` is a distinguished name of a supercombinator and

that the other supercombinator names may be renamed. Then clearly, a Haskell program with supercombinators $x_i\ y_{1,i} \ldots y_{1,m_i} = s_i$ for $i = 1, \ldots, n$ can be expressed in CH as `letrec` $x_1 = \lambda y_{1,1}, \ldots, y_{1,m_1}.s_1; \ldots; x_n = \lambda y_{n,1}, \ldots y_{n,m_n}.s_n$ `in main` and thus we have:

▶ **Corollary 3.18.** *Haskell with only data definitions and renamble supercombinator definitions has a* **GI***-complete alpha-equivalence problem, even if* `letrec`*-expressions are forbidden. Also a rule for garbage collection can be defined such that Theorem 3.13 is applicable.*

In the record calculus [17] records are sets of bindings $l_i \mapsto t_i$ where t_i are expressions of an extended lambda calculus (no `letrec`-bindings). This is like a one-level `letrec`, hence:

▶ **Proposition 3.19.** *The alpha-equivalence problem in the record calculus [17] is* **GI***-complete, provided renaming of the record names is permitted.*

Proof. This follows from Proposition 3.4. ◀

In the record calculus there is no distinguished `main`-label and thus a sufficient rule for garbage collection is not definable in this calculus, i.e. Theorem 3.13 is not applicable.

3.4 Comparison of Types

Another application of our methods is comparison of types under various equivalences, which for example is a useful feature for searching functions with similar types in function libraries [31]. Fix a finite, non-empty set of type constructors tc_i where every type constructor has a fixed arity $ar(tc_i) \in \mathbf{N}_0$. Let α, α_i be type variables of a countably-infinite set of type variables. The syntax of polymorphic types $\tau \in T$ is $\tau, \tau_i \in T ::= \alpha \mid \tau_1 \to \tau_2 \mid tc(\tau_1, \ldots, \tau_{ar(tc)})$. As usual we assume arrow-types to be right-associative. Let similarity, $\sim$, of types be $\sim_1 \circ \sim_2$ where $\sim_1$ allows renaming of type variables and $\sim_2$ is the least congruence on types that respects the axiom $\tau_1 \to \tau_2 \to \tau_3 \sim \tau_2 \to \tau_1 \to \tau_3$. For instance, $(\alpha_1 \to \alpha_2) \to List(\alpha_1) \to List(\alpha_2) \sim List(\alpha_4) \to (\alpha_4 \to \alpha_3) \to List(\alpha_3)$. A type is *positive*, if it is a type variable, or of the form $\tau_1 \to \ldots \to \tau_n$ or $tc(\tau_1, \ldots, \tau_{ar(tc)})$ where every type τ_i does not contain arrow-types.

▶ **Theorem 3.20.** *Similarity of Hindley-Milner polymorphic types (as defined above), even for only positive types, is* **GI***-complete.*

Proof. In **GI** follows using standard methods. **GI**-hardness follows, where the encoding of a directed graph G is as follows: For nodes $v_i, i = 1, \ldots, n$, edges $\{(v_{i_1}, v_{j_1}), \ldots, (v_{i_k}, v_{j_k})\}$, $1 \le i_h, j_h \le n$, let the constructed (positive) type be $v_1 \to \ldots \to v_n \to tc(v_{i_1}, v_{j_1}) \to tc(v_{i_2}, v_{j_2}) \to \ldots \to tc(v_{i_k}, v_{j_k}) \to \alpha$ where v_{i_1}, v_{j_1}, α are now type variables. ◀

4 Structural Congruence in Process Calculi

In this section we consider structural congruence in process calculi. In particular the π-calculus and several fragments are analyzed that not only include an associative and commutative binary operator $\mid$ for parallel composition but also axioms for moving ν-binders, like $\nu x.\nu y.P \equiv \nu y.\nu x.P$ and $(\nu x.P) \mid Q \equiv \nu x.(P \mid Q)$ if $x \notin FV(Q)$.

4.1 Process Calculi with Bindings

We first consider two extended lambda calculi that have a recursive binding scope only at top-level, where bindings may be permutable, and where a structural congruence is defined

extending alpha-equivalence. This covers the call-by-value lambda calculus with futures in [26, 25, 40] and also CHF [34], a concurrent process-extension of Haskell languages.

▶ **Proposition 4.1.** *Deciding structural congruence $\equiv$ in the variants of the lambda-calculus with futures in [25, 40] is* **GI***-complete, as well as of the language CHF modelling Concurrent Haskell with futures [34].*

Proof. **GI**-hardness follows from Proposition 3.4, since these calculi can express renameable, recursive one-level `letrec`-bindings. The proof that deciding structural congruence is in **GI** does not directly follow from results for CH: First one has to verify that a prenex-normal form can be computed, where all ν-binders are on the top of the process, i.e. a process in prenex-normal form is of the form $\nu x_1 \ldots \nu x_n.P$ where P does not contain ν-binders. For encoding these normal forms into graphs, the nested ν-binders are treated like a single binding-operator which binds a *set* of variables, i.e. $\nu X.P$ where $X = \{x_1, \ldots, x_n\}$. Also nested parallel compositions $P_1 \mid P_2 \mid \ldots$ are treated like a (multi-)set of processes $\{P_1, \ldots, P_n\}$. With this preparations the corresponding LDG of a process can be encoded such that isomorphism of the LDGs is equivalent to structural congruence of the processes. ◀

Note that in CHF a garbage collection rule can be defined which is sufficient to apply Theorem 3.13, since there is a distinguished main-thread, while in the lambda calculus with futures there is no such thread and thus Theorem 3.13 is not applicable.

4.2 Structural Congruence in the π-Calculus

We analyze deciding structural congruence in the π-calculus [21, 20, 35]. Note that in the π-calculus a prenex-normal form does not exist, since not every ν-binder can be moved to the top.

Note that in calculi with commutable ν-binders and commutative-associative composition, **GI**-hardness of structural congruence can also be concluded by encoding directed graphs (V, E) with $V = \{v_1, \ldots, v_n\}$ as $\nu v_1, \ldots v_n.P$ where P is a (nested) parallel composition of the components: $c_1(v)$ for any $v \in V$ and $c_2(v, w)$ for every directed edge $(v, w) \in E$ where c_1 is a unary, and c_2 a binary constructor or function symbol. E.g. in the π-calculus with input and output prefixes defined by $\pi ::= x(y) \mid \overline{x}\langle y\rangle$ we can use $\overline{v}\langle w\rangle.0$ for $c_2(v, w)$ and $v(z).0$ for $c_1(v)$ for some name z. These encodings can be found in [14] and are similar to the encodings given in [6]. This shows that structural congruence in several fragments of the π-calculus is at least **GI**-hard. In [14] it was also shown that in the π-calculus with sums and non-renameable defined function symbols (perhaps recursive) deciding structural equivalence is **GI**-complete.

However, in the π-calculus with a replication operator **!** structural congruence includes the axiom $!P \equiv P \mid !P$. With this axiom deciding structural congruence is much harder. We consider the following fragment – called $\mathcal{PIR}$ – which covers several variants of the π-calculus with replication. Let C be an infinite set of constants (atomic actions), then the syntax of $\mathcal{PIR}$ is: $s, s_i \in \mathcal{PIR} := C \mid (s_1 \mid s_2) \mid !s$. We assume that structural congruence is defined by the axioms $(s_1 \mid s_2) \equiv (s_2 \mid s_1)$, $(s_1 \mid (s_2 \mid s_3)) \equiv ((s_1 \mid s_2) \mid s_3)$ and $!s \equiv s \mid !s$.

In [11] it is shown that structural congruence in $\mathcal{PIR}$ can be decided in **EXPSPACE**. Indeed the problem is also **EXPSPACE**-hard, which we show in the following.

First we consider commutative semigroups. Let Σ be an alphabet of constants, written a, b, c, perhaps with indices.

▶ **Lemma 4.2.** *Given equations $s_1 = 1, \ldots, s_n = 1$ where s_i are (commutative) words over Σ, and two further commutative words s, t over Σ, then the decision problem of the word problem $s_1 = 1 \wedge \ldots \wedge s_n = 1 \models s = t$ over commutative monoids is* **EXPSPACE***-complete.*

Proof. Given equations $s_1 = t_1, \ldots, s_n = t_n$ where s_i, t_i are (commutative) words over Σ, and two further commutative words s, t over Σ, then the decision problem of the word problem $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \models s = t$ over commutative semigroups is **EXPSPACE**-complete [19]. We apply this result. Encoding the commutative monoid word problem is easy by adding the axioms for the unit to the given equations. For the other direction we add n fresh constants $g_1, \ldots, g_n$ to the signature and encode the equations as $s_1 g_1 = 1, t_1 g_1 = 1, \ldots, s_n g_n = 1, t_n g_n = 1$. The equation $s_i = s_i g_i t_i = t_i$ for $i = 1, \ldots, n$ is then derivable from these equations. It is also not hard to see that the extension is conservative, i.e., the equational theories are equivalent on words free of $g_1, \ldots g_n$. Since the encodings are polynomial, we are done. ◀

▶ **Theorem 4.3.** *Structural congruence of expressions in $\mathcal{PIR}$ is* **EXPSPACE***-complete.*

Proof. Due to the results of [11] it suffices to show **EXPSPACE**-hardness. For convenience, let us write the expressions without the parallel-operator, and assume that we have commutative words in C^*. Then $w_1!(v_1)\ldots!(v_n) \equiv w_2!(v_1)\ldots!(v_n)$ is equivalent to $v_1 = 1 \wedge \ldots \wedge v_n = 1 \models w_1 = w_2$ in a commutative monoid. Thus Lemma 4.2 implies **EXPSPACE**-hardness. ◀

Since $\mathcal{PIR}$ can be embedded in the π-calculus with replication by simulating the constants with different input-expressions, and omitting ν-binders, we have the following result:

▶ **Corollary 4.4.** *Structural congruence in the π-calculus with replication is* **EXPSPACE***-hard.*

This high complexity is a hint that it is not a good idea to include the replication axiom in the congruence relation. It would be better to include it in the operational semantics and only copy expressions "by need".

Engelfriet and Gelsema [10] investigate variants of the congruence axioms for the replication operator and show decidability of the congruence, however, they do not mention complexity bounds. An application of congruence and complexity of these variants is in [32].

Note that Theorem 4.3 does not apply to variants of congruence for the replication operator as proposed in [10], since the respective congruences are different.

▶ Remark. The complexity (and even the decidability) of the congruence problem for Milner's variant of the π-calculus is still open. We show an example for one of the problematic cases (using term notation) that are not covered by [11]: Let $s = \nu x.(f(x) \mid !(f(x) \mid a)) \mid \nu x.!(g(x) \mid a)$, and $t = \nu x.!(f(x) \mid a) \mid \nu x.(g(x) \mid !(g(x) \mid a))$, then $s \equiv t$ by exchanging the action a and using the replication axiom in both directions.

5 Summary and Conclusion

We have shown that alpha-equivalence of expressions in higher-order core functional programming languages with a recursive let-construct that permits to permute the bindings, and thus also for Haskell-expressions, is **GI**-complete, and consequently, algorithms require exponential time in the worst case. If there is no garbage, then the worst case time complexity is polynomial. This fact allows deduction systems a choice: if the application of algorithms for alpha-equivalence is infeasible in the deduction, then the program transformation (gc) could

be added to the set of transformations, which opens the possibility to use the polynomial time algorithm for garbage free expression as proposed above.

Acknowledgments

We thank the anonymous reviewers for their valuable and helpful comments.

References

1 Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Ann. Pure Appl. Logic*, 117(1-3):95–168, 2002.

2 Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.

3 Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *POPL'95*, pp. 233–246. ACM Press, 1995.

4 Z. M. Ariola and Stefan Blom. Cyclic lambda calculi. In *Proc. 3rd TACS '97*, LNCS 1281, pp. 77–106. Springer, 1997.

5 G. Ausiello, F. Cristiano, and L. Laura. Syntactic isomorphism of CNF Boolean formulas is graph isomorphism complete. *Electronic Colloquium on Computational Complexity*, 19:122, 2012.

6 D. A. Basin. A term equality problem equivalent to graph isomorphism. *Inf. Process. Lett.*, 51(2):61–66, 1994.

7 K. S. Booth and C. J. Colboum. Problems polynomially equivalent to graph isomorphism. Technical Report CS-77-04, University of Waterloo, 1979.

8 C. Calvès and M. Fernández. Matching and alpha-equivalence check for nominal terms. *J. Comput. System Sci.*, 76(5):283–301, 2010.

9 J. Cheney. The complexity of equivariant unification. In *Proc. 31st ICALP*, *LNCS* 3142, pp. 332–344. Springer, 2004.

10 J. Engelfriet and T. Gelsema. A new natural structural congruence in the pi-calculus with replication. *Acta Inf.*, 40(6-7):385–430, 2004.

11 J. Engelfriet and T. Gelsema. An exercise in structural congruence. *Inf. Process. Lett.*, 101(1):1–5, 2007.

12 J. J. Fu. Directed graph pattern matching and topological embedding. *J. Algorithms*, 22(2):372–391, 1997.

13 X. Jiang and H. Bunke. Optimal quadratic-time isomorphism of ordered graphs. *Pattern Recognition*, 32(7):1273–1283, 1999.

14 V. Khomenko and R. Meyer. Checking pi-calculus structural congruence is graph isomorphism complete. In *Proc. 9th ACSD*, pp. 70–79. IEEE Computer Society, 2009.

15 Johannes Köbler. On graph isomorphism for restricted graph classes. In *CiE*, *LNCS* 3988, pp. 241–256. Springer, 2006.

16 J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Comput. Complexity*, 2:301–330, 1992.

17 E. Machkasova. Computational soundness of a call by name calculus of recursively-scoped records. *ENTCS*, 204:147–162, 2008.

18 J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8:275–317, 1998.

19 E. W. Mayr and A. R. Meyer. The complexity of the word problems for commutative semigroups and polynomial ideals. *Adv. in Math.*, 46(3):305–329, 1982.

20 R. Milner. Functions as processes. *Math. Structures Comput. Sci.*, 2(2):119–141, 1992.

21 R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge university press, 1999.

22 A. K. D. Moran. *Call-by-name, call-by-need, and McCarthy's Amb.* PhD thesis, Dept. of Comp. Science, Chalmers university, Sweden, 1998.

23 A. K. D. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. 26th POPL*, pp. 43–56. ACM Press, 1999.

24 A. K. D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.

25 J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *ENTCS*, 173:313–337, 2007.

26 J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, 2006.

27 A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

28 C. Rau, D. Sabel, and M. Schmidt-Schauß. Correctness of program transformations as a termination problem. In *Proc. 6th IJCAR*, LNCS 7364, pp. 462–476. Springer, 2012.

29 C. Rau and M. Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *Proc. 25th UNIF*, pp. 35–41, 2011.

30 C. Rau and M. Schmidt-Schauß. Computing overlappings by unification in the deterministic lambda calculus LR with letrec, case, constructors, seq and variable chains. Frank Report 46, Institut für Informatik. Goethe-Universität Frankfurt, 2011.

31 M. Rittri. Using types as search keys in function libraries. *J. Funct. Program.*, 1(1):71–89, 1991.

32 A. Romanel and C. Priami. On the decidability and complexity of the structural congruence for beta-binders. *Theoret. Comput. Sci.*, 404(1-2):156–169, 2008.

33 D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

34 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *Proc. 13th PPDP*, pp. 101–112. ACM, 2011.

35 D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* Cambridge university press, 2001.

36 M. Schmidt-Schauß and E. Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In *Proc. 19th RTA*, LNCS 5117, pp. 321–335. Springer-Verlag, 2008.

37 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In *Proc. 21st RTA*, LIPIcs 6, pp. 295–310. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

38 M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

39 U. Schöning. Graph isomorphism is in the low hierarchy. *J. Comput. Syst. Sci.*, 37(3):312–323, 1988.

40 J. Schwinghammer, D. Sabel, M. Schmidt-Schauß, and J. Niehren. Correctly translating concurrency primitives. In *Proc. ML '09*, pp. 27–38. ACM, 2009.

41 V.N. Zemlyachenko, N.M. Korneenko, and R.I.Tyshkevich. Graph isomorphism problem. *J. Math. Sci. (N. Y.)*, 29:1426–1481, 1985.

Unification Modulo Nonnested Recursion Schemes via Anchored Semi-Unification

Gert Smolka[1] and Tobias Tebbi[1]

1 **Saarland University, Saarbrücken, Germany**
smolka@ps.uni-saarland.de, ttebbi@ps.uni-saarland.de

Abstract

A recursion scheme is an orthogonal rewriting system with rules of the form $f(x_1, \ldots, x_n) \to s$. We consider terms to be equivalent if they rewrite to the same redex-free possibly infinite term after infinitary rewriting. For the restriction to the nonnested case, where nested redexes are forbidden, we prove the existence of principal unifiers modulo scheme equivalence. We give an algorithm computing principal unifiers by reducing the problem to a novel fragment of semi-unification we call anchored semi-unification. For anchored semi-unification, we develop a decision algorithm that returns a principal semi-unifier in the positive case.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases recursion schemes, semi-unification, infinitary rewriting

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.271

1 Introduction

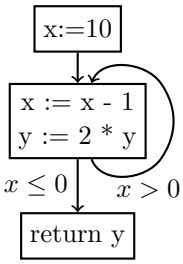
Recursion schemes (in the following shortly called schemes) describe mutually recursive function definitions and go back to the 1970s [18, 2]. They can be understood as rewriting systems with rules of the form $f(x_1, \ldots, x_n) \to s$ such that there is exactly one rule per function symbol f and s is not a redex. Starting with a finite term, the limit of repeatedly rewriting with these rules is a possibly infinite term containing no redexes, see Dershowitz et al. [3]. We consider two terms to be equivalent if they rewrite to the same redex-free term. This equivalence is known as tree equivalence (introduced by Rosen [18]). Given a scheme $\mathcal{S}$, we will speak of $\mathcal{S}$-equivalence in the following. As shown by Courcelle [2], $\mathcal{S}$-equivalence is interreducible to the equivalence of deterministic pushdown automata (DPDA). DPDA equivalence was an open problem for 20 years and was finally shown to be decidable by Sénizergues [21]. Sénizergues' proof yields a non-elementary decision procedure. The best known upper bound for the complexity of the problem is primitive recursive (see Stirling [22]).

In this paper, we will consider schemes without nested redexes, which we call nonnested schemes following Courcelle [2]. Sabelfeld [19] gives a polynomial decision procedure for $\mathcal{S}$-equivalence where $\mathcal{S}$ is a nonnested scheme.

Our motivation to investigate nonnested schemes is compiler verification. Nonnested schemes suffice to encode control flow graphs (CFGs) where everything but register updates and jumps is left uninterpreted. See Figure 1 for an example of a CFG together with a corresponding scheme. If two CFGs result in equivalent recursion schemes, then they are observationally equivalent. So if a compiler optimization produces a transformed CFG whose recursion scheme is equivalent to the recursion scheme of the original CFG, then this optimization is sound. Thus in a verified compiler, a compilation phase that produces a scheme-equivalent CFG can be validated by running a recursion scheme equivalence checker.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 271–286

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$$\begin{array}{lcl} f_1(x,y) & \to & f_2(10,y) \\ f_2(x,y) & \to & \text{if}(x-1>0, f_2(x-1,2*y), f_3(x-1,2*y)) \\ f_3(x,y) & \to & \text{return}(y) \end{array}$$

Figure 1 CFG with corresponding scheme.

We expect that compilation phases like global value numbering, code motion for assignments and register allocation (except for spilling) can be validated with this method.

We solve a more general problem than $\mathcal{S}$-equivalence, namely unification modulo $\mathcal{S}$-equivalence. This relies on our result that $\mathcal{S}$-unification is unitary (i.e., principal $\mathcal{S}$-unifiers exist). Given two terms s and t, we show how to compute a principal $\mathcal{S}$-unifier of s and t if one exists. For our results, we restrict substitutions such that variables are replaced with redex-free terms.

Our method is a certifying algorithm [15] in the following sense. Whenever we compute a principal substitution σ such that σs and σt are $\mathcal{S}$-equivalent, we also have a certificate proving that σs and σt are $\mathcal{S}$-equivalent.

Our method works by reducing the unification problem to a new decidable fragment of semi-unification we call *anchored semi-unification*. Semi-unification is a generalization of ordinary syntactic unification. Semi-unification was first identified by Lankford and Musser [10] in 1978 and rediscovered in different areas ten years later [5, 7, 17]. In its general form, semi-unification was shown to be undecidable by Kfoury et al. [8]. Several decidable fragments of semi-unification are known, but all of them differ significantly from our new fragment (see Section 11). Our semi-unification algorithm can be seen as a restriction of the diverging semi-unification rules of Leiß [12].

2 Overview

Our method is based on a coinductive definition of $\mathcal{S}$-equivalence. We seem to be the first to employ a coinductive definition of $\mathcal{S}$-equivalence. According to our definition, two terms s and t are $\mathcal{S}$-equivalent (i.e., $s \equiv t$) if there is a compliant relation relating s and t. Compared with process equivalence, compliant relations play the role of bisimulations. Our first main result is the existence of principal $\mathcal{S}$-unifiers. A substitution σ is a principal $\mathcal{S}$-unifier of two terms s and t if it is a principal (aka most general) substitution such that $\sigma s \equiv \sigma t$. From the existence of principal $\mathcal{S}$-unifiers, it follows that there are principal pairs. A pair of terms $(f\overline{s}, g\overline{t})$ is a principal pair for the function symbols f and g if for all tuples of terms $\overline{u}$ and $\overline{v}$, we have $f\overline{u} \equiv g\overline{v}$ iff $(f\overline{u}, g\overline{v})$ is a substitution instance of $(f\overline{s}, g\overline{t})$.

By weakening the conditions on compliant relations, we obtain a decidable criterion for finite relations that implies $\mathcal{S}$-equivalence for the pairs in the relation (and their substitution instances). We call finite relations that satisfy this criterion certificates. It turns out that every finite set of principal pairs can be extended to a certificate by adding more principal pairs. Thus there are certificates for all $\mathcal{S}$-equivalences between redexes. On the other hand, a principal certificate contains only principal pairs. A principal certificate is a relation σF where F is a frame and σ is a principal substitution such that σF is a certificate. A frame is roughly a relation on terms of the form $f\overline{x}$ that does not contain a variable twice. We will

show that if there is a principal pair for f and g, then it is possible to compute a frame F and a substitution σ such that σF is a principal certificate containing a principal pair for f and g.

We solve the $\mathcal{S}$-unification problem (ScUP) by reducing it to the anchored semi-unification problem (AnSUP). This is done using three reduction steps.

$$\text{ScUP} \to \text{FIP} \to \text{SUP}^* \to \text{AnSUP}$$

ScUP is the initial problem, asking for a principal $\mathcal{S}$-unifier of two terms $f\overline{s}$ and $g\overline{t}$. To find an $\mathcal{S}$-unifier of $f\overline{s}$ and $g\overline{t}$, it suffices to determine a principal pair for f and g. If there is no principal pair for f and g, then $f\overline{s}$ and $f\overline{g}$ are not $\mathcal{S}$-unifiable. To find a principal pair for f and g, we compute a frame F for f and g such that F can be instantiated to a principal certificate for f and g iff there exists a principal pair for f and g. We call the new problem frame instantiation problem (FIP).

We reduce FIP to the standard semi-unification problem SUP. Since SUP is undecidable in general, we employ a further reduction to the anchored semi-unification problem AnSUP. We show that every instance of SUP obtained by our reduction from FIP translates to an instance of AnSUP. In the diagram above, SUP^* indicates the fragment of SUP reachable by our reduction from FIP. While SUP employs inequalities $s \dot{\preceq} t$, AnSUP employs equations $s \doteq t$ where s and t can contain instance variables αx consisting of a simple variable x and a substitution variable α that represents a substitution. The anchoredness constraint ensures that for every relevant instance variable αx, there is an equation $\alpha x \doteq s$ where s contains no instance variables. We solve instances of AnSUP with terminating semi-unification rules that are a restriction of the rules in [12]. The anchoredness constraint is preserved by applications of the semi-unification rules and allows us to always eliminate instance variables.

The paper is organized as follows. First, we define nonnested schemes and $\mathcal{S}$-equivalence. Then we formulate ScUP and prove the existence of principal $\mathcal{S}$-unifiers and principal pairs. Afterwards, we define FIP with frames and certificates and present the reductions from ScUP to FIP and from FIP to SUP. Next, we define AnSUP and present the reduction from FIP to AnSUP. Finally, we define a solved form and present terminating semi-unification rules for AnSUP. The sections on AnSUP and the semi-unification rules can be read independently of the rest of the paper.

3 Equivalence Modulo Nonnested Schemes

We assume an alphabet of *constants* (ranged over by a, b, c), an alphabet of *function symbols* (ranged over by f, g, h) and an alphabet of *variables* (ranged over by x, y, z). We assume that there are infinitely many variables and finitely many function symbols. We define *terms* (ranged over by s, t, u, v) using the grammar

$$s, t ::= a \mid x \mid s \cdot t \mid f\overline{s}$$

where $\overline{s}$ is a tuple of terms not containing a term of the form $f\overline{t}$.

Note that although we restrict ourselves to a single binary operator $(\cdot)$, we do not loose expressive power compared to full first-order terms since they can be encoded, for example with $a(s_1, s_2, \ldots, s_n) \rightsquigarrow (\cdots((a \cdot s_1) \cdot s_2) \cdot \cdots \cdot s_n)$. We impose the usual discipline that every function symbol has a *fixed arity* and that for $f\overline{s}$, the length of $\overline{s}$ is the arity of f. We omit parentheses such that $s \cdot t \cdot u = s \cdot (t \cdot u)$.

Since we will have a rewrite rule for every function symbol, we call every term of the form $f\overline{s}$ a *redex*. A term is *simple* if it contains no redex. A term is *plain* if it is simple or a redex.

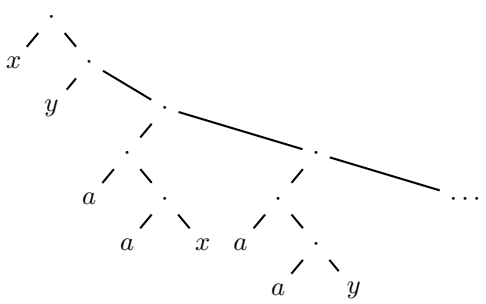
A *substitution* (ranged over by σ, τ, θ) is a function from variables to simple terms. Note that this is a nonstandard restriction and affects our results for unification modulo nonnested schemes. We lift substitutions to range over terms, tuples and sets in the usual way.

A *declaration* of f is a rewrite rule $f\overline{x} \to s_1 \cdot s_2$ with distinct variables $\overline{x}$ and terms s_1, s_2 containing only variables from $\overline{x}$. A *nonnested scheme* (in the following shortly *scheme*) $\mathcal{S}$ is a set of declarations that contains exactly one declaration for every function symbol. We assume that a scheme $\mathcal{S}$ is given. For technical reasons, we require that $\mathcal{S}$ is *reduced*, that is, for every declaration $f\overline{x} \to s_1 \cdot s_2$, both s_1 and s_2 are plain and at least one of them is a redex. Sabelfeld [19] uses a similar restriction of the same name. Given a redex $f\overline{s}$, its *unfolding* $\mathcal{S}(f\overline{s})$ is the term σt where $f\overline{x} \to t$ is the unique declaration of f in $\mathcal{S}$ and σ is a substitution with $\sigma\overline{x} = \overline{s}$. We write $\mathcal{S}_i s$ for t_i where $\mathcal{S}s = t_1 \cdot t_2$ and $i \in \{1, 2\}$.

▶ **Example 1.** For the reduced scheme

$$f(x, y) \to x \cdot g(a \cdot x, y)$$
$$g(x, y) \to y \cdot f(a \cdot x, a \cdot a \cdot y)$$

infinitary rewriting of the redex $f(x, y)$ results in the infinite tree



General Convention. From now on until Section 7, s, t, u and v will always denote plain terms and R will always denote a binary relation on plain terms.

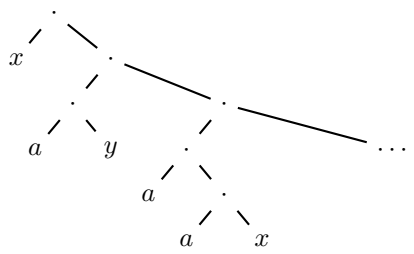
We now give a coinductive definition of $\mathcal{S}$-equivalence. We call a relation R on plain terms *closed* if $R(\mathcal{S}_i s)(\mathcal{S}_i t)$ for all pairs of redexes $(s, t) \in R$ and $i \in \{1, 2\}$. The *kernel* of a relation R is $\mathcal{K}R := \{(s, t) \in R \mid s \text{ or } t \text{ simple}\}$. A relation R is *coreflexive* if $s = t$ whenever Rst. We call a relation R *compliant* if it is closed and $\mathcal{K}R$ is coreflexive. Two plain terms s and t are *$\mathcal{S}$-equivalent* iff there is a compliant relation R with Rst. Since $\mathcal{S}$ is fixed, we simply write $s \equiv t$ to say that s and t are $\mathcal{S}$-equivalent and use $(\equiv)$ to denote the set of all $\mathcal{S}$-equivalent pairs. Note that a compliant relation is essentially a bisimulation between the trees generated by infinitary rewriting.

▶ **Example 2.** For the scheme from Example 1,

$$\{(f(x, a \cdot y), g(y, x)),$$
$$(x, x),\ (g(a \cdot x, a \cdot y), f(a \cdot y, a \cdot a \cdot x)),$$
$$(a \cdot y, a \cdot y),\ (f(a \cdot a \cdot x, a \cdot a \cdot a \cdot y), g(a \cdot a \cdot y, a \cdot a \cdot x)), \ldots\}$$

is a compliant relation containing $(f(x, a \cdot y), g(y, x))$. Observe how this relation corresponds to a walk through the following tree, which can be obtained by infinitary rewriting of either

$f(x, a \cdot y)$ or $g(y, x)$.



▶ **Proposition 3.** *$\mathcal{S}$-equivalence ($\equiv$) is an equivalence relation on plain terms. Moreover, it is the largest compliant relation.*

Proof. Reflexivity follows from the fact that $\{(s, s) \mid s \text{ plain}\}$ is compliant. The other properties follow from the fact that the inverse, union, intersection and composition of compliant relations are compliant. ◀

Our coinductive definition of $\mathcal{S}$-equivalence is equivalent to the usual approach using infinite trees. For every term s, infinitary rewriting with the rules of the scheme yields a possibly infinite redex-free term that can be seen as a possibly infinite binary tree $\mathcal{T}s$ whose internal nodes are labeled with $\cdot$ and whose leaves are labeled with constants and variables. Two terms s and t are tree-equivalent if $\mathcal{T}s = \mathcal{T}t$. We write $(\mathcal{T}s) \cdot (\mathcal{T}t)$ for the tree whose root has exactly $\mathcal{T}s$ and $\mathcal{T}t$ as children. For a formal definition of tree equivalence, see e.g. Courcelle [2]. We will not define $\mathcal{T}s$, but instead we use the following two properties.

1. $\mathcal{T}(s \cdot t) = (\mathcal{T}s) \cdot (\mathcal{T}t)$
2. $\mathcal{T}s = (\mathcal{T}(\mathcal{S}_1 s)) \cdot (\mathcal{T}(\mathcal{S}_2 s))$ if s is a redex

In the following, we write $(\sim_{\mathcal{T}})$ for the relation on plain terms such that $s \sim_{\mathcal{T}} t$ iff $\mathcal{T}s = \mathcal{T}t$. Using property (1.), it is clear that for simple terms s and t, we have $s \sim_{\mathcal{T}} t$ iff $s = t$. For a simple term s, $\mathcal{T}s$ is finite. In contrast to this, $\mathcal{T}s$ is infinite if s is a redex because $\mathcal{S}$ is reduced. Taking these properties together, we obtain that $\mathcal{K}(\sim_{\mathcal{T}})$ is coreflexive. We also have that $(\sim_{\mathcal{T}})$ is closed because if $s \sim_{\mathcal{T}} t$ for redexes s and t, then, by property (2.), $(\mathcal{T}(\mathcal{S}_1 s)) \cdot (\mathcal{T}(\mathcal{S}_2 s)) = (\mathcal{T}(\mathcal{S}_1 t)) \cdot (\mathcal{T}(\mathcal{S}_2 t))$ and hence $\mathcal{S}_i s \sim_{\mathcal{T}} \mathcal{S}_i t$ for $i \in \{1, 2\}$. Thus $(\sim_{\mathcal{T}})$ is a compliant relation and therefore $(\sim_{\mathcal{T}}) \subseteq (\equiv)$.

It remains to show that $(\sim_{\mathcal{T}}) \supseteq (\equiv)$. Assume, for contradiction, that there are plain terms s and t with $s \equiv t$ that are not tree-equivalent. When two trees are different, then they differ at some finite level. Select s and t such that the level l at which $\mathcal{T}s$ and $\mathcal{T}t$ differ is minimal. Since $s \equiv t$, we have that s and t are both redexes because otherwise $s = t$. Since $\mathcal{T}s = (\mathcal{T}(\mathcal{S}_1 s)) \cdot (\mathcal{T}(\mathcal{S}_2 s))$ and $\mathcal{T}t = (\mathcal{T}(\mathcal{S}_1 t)) \cdot (\mathcal{T}(\mathcal{S}_2 t))$ differ at level l, there is $i \in \{1, 2\}$ such that $\mathcal{T}(\mathcal{S}_i s)$ and $\mathcal{T}(\mathcal{S}_i t)$ differ at level $l - 1$. So we have a contradiction because $\mathcal{S}_i s \equiv \mathcal{S}_i t$.

▶ Remark. We can restrict ourselves to plain terms because other terms can be transformed into plain terms by adding additional declarations to the scheme.

Also, the restriction to reduced schemes is inessential because every scheme can be transformed into an equivalent reduced one.

- A declaration $f\overline{x} \to s$ with s simple can be eliminated by replacing every redex $f\overline{t}$ in $\mathcal{S}$ with the simple term $\mathcal{S}(f\overline{t})$.
- A declaration with deep redexes can be split into several declarations, e.g.

$$f(x) \to a \cdot g() \cdot x \quad \rightsquigarrow \quad \begin{array}{l} f(x) \to a \cdot f'(x) \\ f'(x) \to g() \cdot x \end{array}$$

4 Scheme Unification Problem (ScUP)

A substitution σ is an *$\mathcal{S}$-unifier* of two terms s and t if $\sigma s \equiv \sigma t$. We define the usual *instantiation pre-order* ($\preceq$) on tuples of terms and substitutions. For two tuples of terms $\overline{s}$ and $\overline{t}$, we have $\overline{s} \preceq \overline{t}$ if $\sigma\overline{s} = \overline{t}$ for some substitution σ. For two substitutions σ and τ, we have $\sigma \preceq \tau$ if $\tau = \theta \circ \sigma$ for some substitution θ (as usual, $(\theta \circ \sigma)(x) := \theta(\sigma x)$). We call a set Σ of substitutions *quasi-principal* if Σ is either empty or contains a substitution σ such that $\Sigma = \{\tau \mid \sigma \preceq \tau\}$. In this case we call σ a *principal element* of Σ. A *principal $\mathcal{S}$-unifier* of s and t is a principal element of $\{\sigma \mid \sigma s \equiv \sigma t\}$.

▶ **Problem 1** (ScUP)**.** Given two plain terms s and t, find a principal $\mathcal{S}$-unifier of s and t if one exists.

In the remainder of this section, we prove that two terms have a principal $\mathcal{S}$-unifier if they are $\mathcal{S}$-unifiable. The principal $\mathcal{S}$-unifiers will turn out to be essential for our reduction, because they allow us to characterize all $\mathcal{S}$-equivalences between terms with the finite set of principal $\mathcal{S}$-unifiers between terms of the form $f\overline{x}$.

The *closure* $\mathcal{C}(s,t)$ of s and t is the smallest closed relation containing (s,t).

▶ **Proposition 4.** *$s \equiv t$ iff $\mathcal{K}(\mathcal{C}(s,t))$ is coreflexive.*

Proof. If $s \equiv t$, then there is a compliant relation R containing (s,t). Thus $\mathcal{C}(s,t) \subseteq R$. Hence $\mathcal{K}(\mathcal{C}(s,t)) \subseteq \mathcal{K}R$ is coreflexive.

If $\mathcal{K}(\mathcal{C}(s,t))$ is coreflexive, then $\mathcal{C}(s,t)$ is a compliant relation containing (s,t). Thus $s \equiv t$. ◀

▶ **Proposition 5.** *For every redex s, we have $\mathcal{S}(\sigma s) = \sigma(\mathcal{S}s)$.*

▶ **Lemma 6.** $\mathcal{C}(\sigma s, \sigma t) = \sigma\,\mathcal{C}(s,t)$

Proof. Using Proposition 5, we obtain that $\sigma\,\mathcal{C}(s,t)$ is closed. Since also $(\sigma\,\mathcal{C}(s,t))(\sigma s)(\sigma t)$, we have $\mathcal{C}(\sigma s, \sigma t) \subseteq \sigma\,\mathcal{C}(s,t)$. It remains to show that $\mathcal{C}(\sigma s, \sigma t) \supseteq \sigma\,\mathcal{C}(s,t)$. This follows from a simple induction on the derivations of the elements in $\mathcal{C}(s,t)$. ◀

The following well-known result generalizes the existence of principal unifiers for ordinary unification to infinite systems of equations. Note that σR is coreflexive iff σ is an (ordinary) unifier of R.

▶ **Proposition 7.** *For a relation R on terms with finitely many variables, $\{\sigma \mid \sigma R$ coreflexive$\}$ is quasi-principal.*

Proof. See Proposition 4.10 in Eder [4]. ◀

▶ **Theorem 8.** *$\{\sigma \mid \sigma s \equiv \sigma t\}$ is quasi-principal.*

Proof. We have

$$
\begin{aligned}
&\{\sigma \mid \sigma s \equiv \sigma t\} \\
&= \{\sigma \mid \mathcal{K}(\mathcal{C}(\sigma s, \sigma t)) \text{ coreflexive}\} && \text{by Proposition 4} \\
&= \{\sigma \mid \mathcal{K}(\sigma\,\mathcal{C}(s,t)) \text{ coreflexive}\} && \text{by Lemma 6} \\
&= \{\sigma \mid \sigma(\mathcal{K}(\mathcal{C}(s,t))) \text{ coreflexive}\}
\end{aligned}
$$

and $\{\sigma \mid \sigma(\mathcal{K}(\mathcal{C}(s,t))) \text{ coreflexive}\}$ is quasi-principal by Proposition 7. ◀

For a relation R on plain terms, we define $\uparrow R := \{(s',t') \mid \exists (s,t) \in R.\ (s,t) \preceq (s',t')\}$. Given function symbols f and g, we call $(f\overline{s}, g\overline{t})$ a *principal pair* for (f,g) if $\uparrow\{(f\overline{s}, g\overline{t})\} = \{(f\overline{u}, g\overline{v}) \mid f\overline{u} \equiv g\overline{v}\}$. Note that a principal pair for f and g completely characterizes $\mathcal{S}$-equivalence for terms of the form $f\overline{s}$ and $g\overline{t}$.

▶ **Corollary 9.** *If σ is a principal $\mathcal{S}$-unifier of $f\overline{x}$ and $g\overline{y}$, and $\overline{x}, \overline{y}$ are pairwise distinct variables, then $\sigma(f\overline{x}, g\overline{y})$ is a principal pair.*

5 Frame Instantiation Problem (FIP)

In this section, we introduce the frame instantiation problem, which will allow us to compute principal pairs. In Section 6, we will then show how to construct a frame that can be instantiated to a certificate consisting of principal pairs only.

For a relation R on redexes, we define $\Uparrow R := \uparrow R \cup \{(s,s) \mid s \text{ simple}\}$. A finite relation R on redexes is a *certificate* if $\Uparrow R(\mathcal{S}_i s)(\mathcal{S}_i t)$ for all redexes $(s,t) \in R$ and $i \in \{1,2\}$. Note that a certificate is essentially a bisimulation up-to instantiation in the sense of up-to techniques for bisimulations [20].

▶ **Example 10.** For the scheme from Example 1,

$$R := \{(f(x, a \cdot y), g(y,x)),\ (g(y,x), f(x, a \cdot y))\}$$

is a certificate. Observe that $\Uparrow R$ is a superset of the compliant relation from Example 2. Also note that R consists of two principal pairs.

▶ **Lemma 11.** *Let R be a certificate. Then $\Uparrow R \subseteq (\equiv)$.*

Proof. It suffices to show that $\Uparrow R$ is compliant. We have that $\mathcal{K}\Uparrow R$ is coreflexive because $\mathcal{K}\Uparrow R = \Uparrow(\mathcal{K}R) = \Uparrow\{\}$. It remains to show that $\Uparrow R$ is closed. Consider a pair of redexes $(s,t) \in \Uparrow R$. We have to show that $\Uparrow R(\mathcal{S}_i s)(\mathcal{S}_i t)$ for $i \in \{1,2\}$. By the definition of $\Uparrow R$, there are redexes $(u,v) \in R$ with $(u,v) \preceq (s,t)$. Since R is a certificate, we have $\Uparrow R(\mathcal{S}_i u)(\mathcal{S}_i v)$. As $(\mathcal{S}_i u, \mathcal{S}_i v) \preceq (\mathcal{S}_i s, \mathcal{S}_i t)$ by Proposition 5, it follows that $\Uparrow R(\mathcal{S}_i s)(\mathcal{S}_i t)$. ◀

We write $s \approx t$ if s and t are redexes with the same function symbol, and $(s,t) \approx (u,v)$ if $s \approx u$ and $t \approx v$. We call a relation R on redexes *unitary* if there are no distinct pairs $(s_1,s_2), (t_1,t_2) \in R$ with $(s_1,s_2) \approx (t_1,t_2)$. Note that every unitary relation is finite because there are only finitely many function symbols. We call a unitary relation F on redexes of the form $f\overline{x}$ a *frame* if no variable occurs twice in F.

▶ **Example 12.** Consider the frame $F := \{(f(x_1,x_2), g(x_3,x_4)),\ (g(x_5,x_6), f(x_7,x_8))\}$. There is a substitution σ such that σF is the certificate from Example 10. Moreover, σ is principal with this property if it is the identity on all variables not occurring in F.

▶ **Problem 2** (FIP)**.** Given a frame F, find a principal element of $\{\sigma \mid \sigma F \text{ is certificate}\}$ if it exists.

It is easy to decide if a finite relation is a certificate. Thus a certificate proves that its elements are $\mathcal{S}$-equivalent. So for every solution σ of a frame F, we obtain a certificate (in the sense of a certifying algorithm [15]) for the fact that all pairs of terms in σF are $\mathcal{S}$-equivalent.

6 ScUP to FIP

To solve ScUP, it suffices to compute principal pairs. Suppose we want to compute a principal $\mathcal{S}$-unifier of two plain terms s and t. We distinguish three cases.

1. If s and t are both redexes, then consider a principal pair (s', t') with $(s', t') \approx (s, t)$ and fresh variables. We obtain an ordinary unification problem because
$$\{\sigma \mid \sigma s \equiv \sigma t\} = \{\sigma \mid (s', t') \preceq \sigma(s, t)\} = \{\sigma \mid \exists \tau.\ \tau(s', t') = \sigma(s, t)\}$$
2. If s and t are both simple, then we are about to solve an ordinary unification problem since for simple terms, $\mathcal{S}$-equivalence is (syntactic) equality.
3. Otherwise, one of them is a redex and the other one is a simple term and hence they cannot be $\mathcal{S}$-unifiable.

So in the following, we only consider the first case and construct a frame F such that for its principal solution σ, a suitable principle pair is contained in σF.

We call a relation R on redexes *pointwise $\mathcal{S}$-unifiable* if s and t are $\mathcal{S}$-unifiable whenever Rst. We call a relation R on redexes *function-closed* if $\Uparrow R(\mathcal{S}_i s)(\mathcal{S}_i t)$ whenever Rst, $i \in \{1, 2\}$ and $\mathcal{S}_i s$ and $\mathcal{S}_i t$ are both redexes. For example, the frame from Example 12 is pointwise $\mathcal{S}$-unifiable and function-closed.

▶ **Lemma 13.** *For every function-closed and pointwise $\mathcal{S}$-unifiable frame F, there is a substitution σ such that σF is a certificate containing only principal pairs.*

Proof. For $(s, t) \in F$, let $\sigma_{s,t}$ be a principal $\mathcal{S}$-unifier of s and t. These $\mathcal{S}$-unifiers exist because F is pointwise $\mathcal{S}$-unifiable. Since the elements of F have disjoint variables, the following substitution is well-defined.

$$\tau x = \begin{cases} \sigma_{s,t} x & \text{if there is } (s, t) \in F \text{ s.t. } x \text{ occurs in } (s, t) \\ x & \text{otherwise} \end{cases}$$

For all $(s, t) \in F$, we have that $\sigma_{s,t}(s, t) = \tau(s, t)$ and hence $\tau(s, t)$ is a principal pair by Corollary 9. Thus τF consists only of principal pairs and $\Uparrow(\tau F) = (\equiv) \cap \Uparrow F$.

It remains to show that τF is a certificate. Consider a pair of redexes $(s, t) \in \tau F$. We have to show that $\Uparrow(\tau F)(\mathcal{S}_i s)(\mathcal{S}_i t)$ for $i \in \{1, 2\}$. There are $(s', t') \in F$ with $\tau(s', t') = (s, t)$. We have that $s \equiv t$ and hence $\mathcal{S}_i s \equiv \mathcal{S}_i t$. By Proposition 5, $\tau(\mathcal{S}_i s') = \mathcal{S}_i s \equiv \mathcal{S}_i t = \tau(\mathcal{S}_i t')$. Thus either both $\mathcal{S}_i s'$ and $\mathcal{S}_i t'$ are redexes or both are simple. Hence and because F is function-closed, we have $\Uparrow F(\mathcal{S}_i s')(\mathcal{S}_i t')$. Thus also $\Uparrow F(\tau(\mathcal{S}_i s'))(\tau(\mathcal{S}_i t'))$ and equivalently $\Uparrow F(\mathcal{S}_i s)(\mathcal{S}_i t)$. Since $\Uparrow(\tau F) = (\equiv) \cap \Uparrow F$, we conclude that $\Uparrow(\tau F)(\mathcal{S}_i s)(\mathcal{S}_i t)$. ◀

However, we are still missing a way to construct an appropriate function-closed and pointwise $\mathcal{S}$-unifiable frame. It turns out that we get pointwise $\mathcal{S}$-unifiability for free if we just make the frame as small as we can.

Given function symbols f and g, we write $\mathcal{F}(f, g)$ for a fixed function-closed frame with minimum cardinality containing a pair $(f\overline{x}, g\overline{y})$ for some pairwise distinct variables $\overline{x}, \overline{y}$. Since every maximum cardinality frame is function-closed, $\mathcal{F}(f, g)$ always exists. For example, we can take the frame from Example 12 for $\mathcal{F}(f, g)$ where f and g are the function symbols from Example 1. One can easily compute $\mathcal{F}(f, g)$ by incrementally adding elements $(h\overline{x}, h'\overline{y})$ with fresh and distinct variables $\overline{x}, \overline{y}$ as required by the function-closedness constraint.

▶ **Lemma 14.** *If $f\overline{s}$ and $g\overline{t}$ are $\mathcal{S}$-unifiable, then $\mathcal{F}(f, g)$ is pointwise $\mathcal{S}$-unifiable.*

Proof. Suppose, for contradiction, that $F := \{(u,v) \in \mathcal{F}(f,g) \mid u, v\ \mathcal{S}\text{-unifiable}\} \neq \mathcal{F}(f,g)$. Then F has strictly smaller cardinality. We have $\Uparrow F(f\overline{s})(g\overline{t})$ since $f\overline{s}$ and $g\overline{t}$ are $\mathcal{S}$-unifiable. Thus we can conclude a contradiction by showing that F is also function-closed. Let $(u,v) \in F$ and $i \in \{1,2\}$ such that $\mathcal{S}_i u$ and $\mathcal{S}_i v$ are both redexes. We need to show that $\Uparrow F(\mathcal{S}_i u)(\mathcal{S}_i v)$. Since $\mathcal{F}(f,g)$ is function-closed, we have $(\Uparrow \mathcal{F}(f,g))(\mathcal{S}_i u)(\mathcal{S}_i v)$. Select a substitution σ with $\sigma u \equiv \sigma v$. By Proposition 5, we have $\sigma(\mathcal{S}_i u) = \mathcal{S}_i(\sigma u) \equiv \mathcal{S}_i(\sigma v) = \sigma(\mathcal{S}_i v)$. Thus $\mathcal{S}_i u$ and $\mathcal{S}_i v$ are $\mathcal{S}$-unifiable and hence $\Uparrow F(\mathcal{S}_i u)(\mathcal{S}_i v)$. ◀

▶ **Corollary 15.** *If $f\overline{s}$ and $g\overline{t}$ are $\mathcal{S}$-unifiable, then $\{\sigma \mid \sigma\,\mathcal{F}(f,g)$ is certificate$\}$ is nonempty and for every principal element σ, we have that $\sigma\,\mathcal{F}(f,g)$ contains a principal pair for (f,g).*

We will reduce FIP to AnSUP in a way that a frame F is mapped to an anchored system of equations E with $\{\sigma \mid \sigma F$ is certificate$\} = \{\sigma \mid \sigma$ semi-unifies $E\}$ and we will prove that $\{\sigma \mid \sigma$ semi-unifies $E\}$ is quasi-principal. Thus if $f\overline{s}$ and $g\overline{t}$ are $\mathcal{S}$-unifiable, then $\{\sigma \mid \sigma\,\mathcal{F}(f,g)$ is certificate$\}$ contains a principal element σ and $\sigma\,\mathcal{F}(f,g)$ contains a principal pair for (f,g). Otherwise $\{\sigma \mid \sigma\,\mathcal{F}(f,g)$ is certificate$\}$ is empty. This concludes the reduction from ScUP to FIP.

7 FIP to SUP

In this section, we define the semi-unification problem (SUP) using systems of inequalities and give a reduction from FIP to SUP.

Let D be a relation on tuples of plain terms. We write the elements $(\overline{s},\overline{t}) \in D$ as *inequalities* $\overline{s} \dot{\preceq} \overline{t}$ and call D a *system of inequalities*. We call D *directed* if $\overline{s} \preceq \overline{t}$ for all $(\overline{s} \dot{\preceq} \overline{t}) \in D$. If σD is directed, then σ is a *semi-unifier* of D.

▶ **Problem 3** (SUP)**.** Given a system of inequalities D, find a principal semi-unifier of D.

In this section, we will show how to construct a system of inequalities with the same principal solution as a given instance of FIP.

Note that our definition of SUP corresponds roughly to the usual definition of the semi-unification problem (e.g. in Henglein [5]). Since semi-unification is undecidable, we cannot hope to solve the problem in general. Instead, we only consider the image of the reduction from FIP. In Section 9, we will show how to reduce this image to the anchored fragment.

For a unitary relation R, we define the *projection* of a pair (s,t) of plain terms as follows.

$$\pi_R(s,t) := \begin{cases} (s',t') & \text{if } Rs't' \text{ and } (s',t') \approx (s,t) \\ (s,s) & \text{if } s,t \text{ simple} \\ () & \text{otherwise} \end{cases}$$

Note that $\pi_R(s,t)$ is well-defined because R is unitary. The following proposition holds due to the fact that $(s,s) \preceq (s,t)$ iff $s = t$.

▶ **Proposition 16.** *Let R be unitary. Then $\Uparrow Rst$ iff $\pi_R(s,t) \preceq (s,t)$.*

For a unitary relation R, we construct the system of inequalities

$$\mathcal{D}\,R := \{\pi_R(\mathcal{S}_i s, \mathcal{S}_i t) \dot{\preceq} (\mathcal{S}_i s, \mathcal{S}_i t) \mid Rst, i \in \{1,2\}\}$$

▶ **Example 17.** For the frame F from Example 12, we construct the system of inequalities

$$\begin{aligned}\mathcal{D}\,F = \{ \quad & (x_1, x_1) \dot{\preceq} (x_1, x_4),\\ & (g(x_5, x_6), f(x_7, x_8)) \dot{\preceq} (g(a \cdot x_1, x_2), f(a \cdot x_3, a \cdot a \cdot x_4)),\\ & (x_6, x_6) \dot{\preceq} (x_6, x_7),\\ & (f(x_1, x_2), g(x_3, x_4)) \dot{\preceq} (f(a \cdot x_5, a \cdot a \cdot x_6), g(a \cdot x_7, x_8)) \quad \}\end{aligned}$$

▶ **Proposition 18.** *Let R be unitary. Then $\mathcal{D}\,R$ is directed iff R is a certificate.*

Proof. Follows from the construction of $\mathcal{D}\,R$ using Proposition 16. ◀

▶ **Proposition 19.** *Let R be unitary. Then $\sigma(\pi_R(s,t)) = \pi_{(\sigma R)}(\sigma s, \sigma t)$ and $\sigma(\mathcal{D}\,R) = \mathcal{D}(\sigma R)$.*

▶ **Lemma 20.** *Let R be unitary. Then σ semi-unifies $\mathcal{D}\,R$ iff σR is a certificate.*

Proof. Follows immediately from Proposition 18 and Proposition 19. ◀

Thus $\mathcal{D}$ is a reduction from FIP to SUP.

8 Anchored Semi-Unification Problem (AnSUP)

The definition as well as the algorithm for AnSUP rely on a formulation of semi-unification that uses systems of equations between terms with explicit substitution variables instead of inequalities. A similar formulation has been used by Leiß [12]. For example, the inequalities $(x_1, x_2) \dot{\preceq} (x_3, x_4)$ and $(x_5) \dot{\preceq} (x_6)$ corresponds to the equations $\alpha x_1 \doteq x_3$, $\alpha x_2 \doteq x_4$ and $\beta x_5 \doteq x_6$ with the substitution variables α and β. The substitution variables make the additional substitution on the left-hand side of inequalities explicit.

We call the variables we have used so far *simple variables* and assume an additional alphabet of *substitution variables* (ranged over by α, β, γ). An *instance variable* αx is a pair of a substitution variable and a simple variable. We define a new kind of *terms* that extend the simple terms we defined before with instance variables.

$$s, t ::= a \mid x \mid s \cdot t \mid \alpha x$$

Given a simple term s and a substitution variable α, we write $\hat{\alpha} s$ for the term obtained from s by replacing every variable x by αx. As before, substitutions are functions from variables to simple terms. In the semi-unification context [12, 5, 7], this is the standard notion of substitution. We lift substitutions to terms according to the equations $\sigma(\alpha x) = \hat{\alpha}(\sigma x)$, $\sigma(s \cdot t) = \sigma s \cdot \sigma t$ and $\sigma a = a$.

We say a term s *occurs* in a term t if s is a subterm of t. We do not consider x to be a subterm of αx.

An *assignment* A is a function from substitution variables to substitutions. Assignments are lifted to terms such that As is the term obtained from s by replacing every occurrence of an instance variable αx with the simple term $(A\alpha)x$.

A *system of equations* E is a finite set of pairs of terms. We take the freedom to write $s \doteq t$ for a pair (s, t). A substitution σ is a *semi-unifier* of E if there is an assignment A such that for all $s \doteq t \in E$, we have $A(\sigma s) = A(\sigma t)$.

The formulation of semi-unification as just presented has the same expressivity as the formulation with inequalities. We will now restrict the systems of equations we consider to obtain the anchored fragment.

An *atom* (ranged over by X, Y, Z) is either a simple variable or an instance variable. A *partial equivalence relation* (*PER*) is a symmetric and transitive relation. Note that a PER ($\sim$) is an equivalence relation on $\{X \mid X \sim X\}$.

The anchoredness condition ensures that there is an equation $\alpha x \doteq s$ with s simple for every instance variable αx that might occur when the semi-unification rules to be defined later are applied. A system of equations E is *anchored* with a PER ($\sim$) on atoms if the following conditions hold.

1. If $s \doteq t \in E$ with X and Y occurring in $s \doteq t$, then $X \sim Y$.
2. If $x \sim y$ and $\alpha x \sim \alpha x$, then $\alpha x \sim \alpha y$.
3. If $\alpha x \sim \alpha x$, then there is a simple term s with $\alpha x \doteq s \in E$ or $x \doteq s \in E$.

For example, $\{x \doteq z \cdot \beta y,\ \alpha x = a \cdot b\}$ is not anchored, but $\{x \doteq z \cdot \beta y,\ \alpha x = a \cdot b,\ \beta y = z,\ \alpha z = z\}$ is anchored with the symmetric and transitive closure of $\{(x, z), (z, \beta y), (\alpha x, \alpha z), (\alpha z, z)\}$.

▶ **Problem 4** (AnSUP)**.** Given an anchored system of equations E, find a principal semi-unifier.

For comparison to other fragments of semi-unification, we now give a definition of the anchoredness condition for systems of inequalities. A system of inequalities is anchored if there is a partition X of the variables such that for every inequality $\overline{s} \dot{\preceq} \overline{t}$, the following two conditions hold.

1. All variables in $\overline{t}$ are in a single class of the partition X.
2. All variables in $\overline{s}$ are in a single class A of the partition X and every variable in A occurs at least once at a position in $\overline{s}$ that also exists in $\overline{t}$. Expressed formally, there is a tuple $\overline{u}$ and substitutions σ, τ such that $\sigma\overline{u} = \overline{s}$, $\tau\overline{u} = \overline{t}$ and for all variables $x \in A$, we have that $\sigma x = x$ and x occurs in u.

9 FIP to AnSUP

Let F be a frame. We will translate the system of inequalities $\mathcal{D}\,F$ into an equivalent anchored system of equations E such that σ semi-unifies E iff σ semi-unifies $\mathcal{D}\,F$.

The case where $\mathcal{D}\,F$ contains an element of the form $() \dot{\preceq} (s, t)$ is trivial, since in this case, there is no substitution that semi-unifies $\mathcal{D}\,F$. So in the following, we assume that $\mathcal{D}\,F$ contains only inequalities between pairs.

We construct E by translating every element of $\mathcal{D}\,F$ into equations according to the rules

$$(f\overline{x}, g\overline{y}) \dot{\preceq} (f\overline{s}, g\overline{t}) \quad \rightsquigarrow \quad \alpha\overline{x} \doteq \overline{s}, \alpha\overline{y} \doteq \overline{t}$$
$$(s, s) \dot{\preceq} (s, t) \quad \rightsquigarrow \quad s \doteq t$$

where α is a fresh substitution variable for every inequality and $\alpha\overline{x} \doteq \overline{s}$ stands for $\alpha x_1 \doteq s_1, \ldots, \alpha x_n \doteq s_n$ with $\overline{x} = (x_1, \ldots, x_n)$ being a tuple of variables and $\overline{s} = (s_1, \ldots, s_n)$ being a tuple of simple terms of the same length. Note that the rules cover all elements of $\mathcal{D}\,F$ and that in the second rule, s and t are always simple terms.

▶ **Example 21.** The system of inequalities from Example 17 is translated into the system of equations

$$x_1 \doteq x_4,$$
$$\alpha x_5 \doteq a \cdot x_1,\ \alpha x_6 \doteq x_2, \qquad \alpha x_7 \doteq a \cdot x_3,\ \alpha x_8 \doteq a \cdot a \cdot x_4,$$
$$x_6 \doteq x_7,$$
$$\beta x_1 \doteq a \cdot x_5,\ \beta x_2 \doteq a \cdot a \cdot x_6,\ \beta x_3 \doteq a \cdot x_7,\ \beta x_4 \doteq x_8$$

▶ **Lemma 22.** *Let E be a translation of $\mathcal{D}\,F$ for some frame F. Then σ semi-unifies $\mathcal{D}\,F$ iff σ semi-unifies E.*

Proof. Since distinct inequalities in $\mathcal{D}\,F$ are translated into equations with disjoint substitution variables, it suffices to consider the translation E' of a singleton subset $D' \subseteq \mathcal{D}\,F$.

Let $D' = \{(f\overline{x}, g\overline{y}) \dot{\preceq} (f\overline{s}, g\overline{t})\}$. Then $E' = \{\alpha\overline{x} \doteq \overline{s}, \alpha\overline{y} \doteq \overline{t}\}$. If σ semi-unifies D', then there is a substitution τ with $\tau(\sigma(f\overline{x}, g\overline{y})) = \sigma(f\overline{s}, g\overline{t})$. Thus σ semi-unifies E' with the assignment A where $A\alpha := \tau$. If σ semi-unifies E', then σ semi-unifies D' because $(A\alpha)(\sigma\overline{x}) = A(\sigma(\alpha\overline{x})) = A(\sigma\overline{s}) = \sigma\overline{s}$ and the same for $\alpha\overline{y}$ and $\overline{t}$.

Let $D' = \{(s, s) \dot{\preceq} (s, t)\}$ with s and t being simple. Then $E' = \{s \doteq t\}$. If σ semi-unifies D', then there is a substitution τ with $\tau(\sigma s) = \sigma s$ and $\tau(\sigma s) = \sigma t$. Thus $\sigma s = \sigma t$ and hence σ semi-unifies E'. If σ semi-unifies E', then $\sigma s = \sigma t$ and hence σ semi-unifies D'. ◀

We need the following technical lemma to show that the translation is anchored.

▶ **Lemma 23.** *Let $(\sim)$ be an equivalence relation on simple variables and let E be a system of equations containing only equations of the form $\alpha x \doteq s$ or $s \doteq t$ where s and t are simple terms. Then E is anchored if the following conditions are satisfied.*

1. *If αx occurs in E and $x \sim y$, then αy occurs in E.*
2. *If $\alpha x_1 \doteq s \in E$ and $\alpha x_2 \doteq t \in E$, then $y_1 \sim y_2$ for all simple variables y_1, y_2 that occur in s or t.*
3. *If $s \doteq t \in E$, then $x \sim y$ for all simple variables x, y that occur in s or t.*

Proof. Let $\hat{\sim}$ be the smallest symmetric relation on atoms such that

- If $x \sim y$, then $x \mathrel{\hat{\sim}} y$.
- If αx and αy occur in E, then $\alpha x \mathrel{\hat{\sim}} \alpha y$.
- If $\alpha x \doteq s \in E$ and y occurs in s, then $\alpha x \mathrel{\hat{\sim}} y$.

The transitive closure $\hat{\sim}^*$ of $\hat{\sim}$ is a PER. We will show that E is anchored with $\hat{\sim}^*$. But first, we prove that $x \sim y$ whenever $x \mathrel{\hat{\sim}^*} y$. We do this by induction on the length of the shortest sequence $x \mathrel{\hat{\sim}} X_1 \mathrel{\hat{\sim}} \cdots \mathrel{\hat{\sim}} X_n \mathrel{\hat{\sim}} y$. If $n = 0$, then the claim follows immediately from the definition of $\hat{\sim}$. For the inductive case, we distinguish two cases. If there is some simple variable X_i with $i \in \{1, \ldots, n\}$, then the inductive hypothesis for $x \mathrel{\hat{\sim}} X_1 \mathrel{\hat{\sim}} \cdots \mathrel{\hat{\sim}} X_i$ and $X_i \mathrel{\hat{\sim}} \cdots \mathrel{\hat{\sim}} X_n \mathrel{\hat{\sim}} y$ yield $x \sim X_i$ and $X_i \sim y$ and thus $x \sim y$. Otherwise, by the definition of $\hat{\sim}$, there are simple variables $z_1, \ldots, z_n$ and a single substitution variable α such that $X_i = \alpha z_i$ for all $i \in \{1, \ldots, n\}$. By the definition of $\hat{\sim}$, there are simple terms s, t such that $\alpha z_1 \doteq s \in E$ and $\alpha z_n \doteq t \in E$ where x occurs in s and y occurs in t. Thus $x \sim y$ by requirement (2) on $(\sim)$.

Now, we show that E is anchored with $\hat{\sim}^*$.

1. Let $s \doteq t \in E$ with X and Y occurring in $s \doteq t$. We need to show that $X \mathrel{\hat{\sim}^*} Y$. If s and t are simple, then this follows immediately from requirement (3) on $(\sim)$. Otherwise $s = \alpha x$ for some instance variable αx and the claim follows from the definition of $\hat{\sim}$.
2. Let $x \mathrel{\hat{\sim}^*} y$ and $\alpha x \mathrel{\hat{\sim}^*} \alpha x$. We need to show that $\alpha x \mathrel{\hat{\sim}^*} \alpha y$. Because of requirement (1) on $(\sim)$, it suffices to show that $x \sim y$. As we proved before, this follows from $x \mathrel{\hat{\sim}^*} y$.
3. Let $\alpha x \mathrel{\hat{\sim}^*} \alpha x$. It suffices to show that there is a simple term s with $\alpha x \doteq s \in E$. This follows from the conditions on E as $\hat{\sim}^*$ only contains instance variables from E. ◀

▶ **Lemma 24.** *Let E be a translation of $\mathcal{D}\,F$ for some frame F. Then E is anchored.*

Proof. Consider the equivalence relation $(\sim)$ on simple variables such that $x \sim y$ iff x and y occur in the same element of F or $x = y$. It is easy to check that $(\sim)$ and E satisfy the conditions of Lemma 23. ◀

10 Solving AnSUP

In this section, we present rules to solve the anchored semi-unification problem. Our rules are a restriction of the rules presented by Leiß [12].

We first define the solved form computed by our algorithm. We write $E, s \doteq t$ for the disjoint union $E \dot\cup \{s \doteq t\}$. Given a system of equations E, we call an atom X *eliminated* if E has the form $E', X \doteq s$ such that neither X nor (in case X is a simple variable) an instance variable αX occur in E' or in s. A system of equations E is in *solved form* if all equations have the form $X \doteq s$ where X is eliminated and s is simple.

▶ **Lemma 25.** *If a system of equations E is in solved form, then it has a principal semi-unifier.*

Proof. Since E is in solved form, we can unambiguously define a substitution σ and an assignment A as follows.

$$\sigma x = \begin{cases} s & \text{if } x \doteq s \in E \\ x & \text{else} \end{cases}$$

$$(A\alpha)x = \begin{cases} s & \text{if } \alpha x \doteq s \in E \\ x & \text{else} \end{cases}$$

For σ to be a semi-unifier of E, we have to show that for every equation $X \doteq s \in E$ we have $A(\sigma X) = A(\sigma s)$. Since E is in solved form, s is simple and cannot contain an eliminated simple variable. Thus $A(\sigma s) = As = s$. So it suffices to show that $A(\sigma X) = s$.

If $X = x$ for some simple variable x, then $\sigma x = s$ by the definition of σ. Since s is simple, it follows that $A(\sigma x) = A(s) = s$.

If $X = \alpha x$ for some instance variable αx, then $\sigma x = x$ because otherwise x would be eliminated and hence αx could not occur in E. So it follows that $A(\sigma(\alpha x)) = A(\alpha x) = s$.

It remains to show that σ is principal. Consider some other semi-unifier τ of E. Since τ unifies all equations of the form $x \doteq s \in E$, we have that $\tau x = \tau(\sigma x)$ for all simple variables x. Thus $\sigma \preceq \tau$. ◀

A system of equations E is *clashed* if one of the following conditions holds.

- $a \doteq s \in E$ or $s \doteq a \in E$ where $a \neq s$ and s is not an atom.
- $x \doteq s \in E$ where x or αx for some α occurs in s and s is not an atom.
- $\alpha x \doteq s \in E$ where αx occurs in $s \neq \alpha x$.

▶ **Proposition 26.** *A clashed system of equations has no semi-unifier.*

The rules in Figure 2 transform every anchored system of equations into an equivalent system of equations that is clashed or in solved form. To ensure termination, the rules must not be applied to a clashed system of equations. We write $s[t/x]$ for σs where σ is the unique substitution satisfying $\sigma x = t$ and $\sigma y = y$ for all y with $y \neq x$. We write $s[t/\alpha x]$ for the term that is obtained from s by replacing every occurrence of αx with t. We write $E[s/X]$ for $\{t[s/X] \doteq u[s/X] \mid t \doteq u \in E\}$.

$$
\begin{array}{lrcl}
R_{\text{bop}} & E, s_1 \cdot s_2 \doteq t_1 \cdot t_2 & \Longrightarrow & E, s_1 \doteq t_1, s_2 \doteq t_2 \\
R_{\text{refl}} & E, s \doteq s & \Longrightarrow & E \\
R_{\text{orient1}} & E, s \doteq \alpha x & \Longrightarrow & E, \alpha x \doteq s \text{ if } s \text{ is not an instance variable} \\
R_{\text{orient2}} & E, s \doteq x & \Longrightarrow & E, x \doteq s \text{ if } s \text{ is not an atom} \\
R_{\text{elim1}} & E, \alpha x \doteq s & \Longrightarrow & E[s/\alpha x], \alpha x \doteq s \text{ if } s \text{ is simple} \\
R_{\text{elim2}} & E, x \doteq s & \Longrightarrow & E[s/x], x \doteq s \text{ if } x \text{ does not occur in } s \text{ and } s \text{ is simple}
\end{array}
$$

Figure 2 Semi-Unification Rules.

▶ **Example 27.** We transform the following anchored system of equations into solved form.

$$
\begin{array}{rl}
 & \alpha x_1 \doteq a \cdot x_3,\ \alpha x_2 \doteq a \cdot (a \cdot x_4),\ \alpha x_3 \doteq a \cdot x_1,\ \alpha x_4 \doteq x_2,\ x_1 \doteq x_4 \\
\overset{R_{\text{elim2}}}{\Longrightarrow} & \alpha x_4 \doteq a \cdot x_3,\ \alpha x_2 \doteq a \cdot (a \cdot x_4),\ \alpha x_3 \doteq a \cdot x_4,\ \alpha x_4 \doteq x_2,\ x_1 \doteq x_4 \\
\overset{R_{\text{elim1}}}{\Longrightarrow} & x_2 \doteq a \cdot x_3,\ \alpha x_2 \doteq a \cdot (a \cdot x_4),\ \alpha x_3 \doteq a \cdot x_4,\ \alpha x_4 \doteq x_2,\ x_1 \doteq x_4 \\
\overset{R_{\text{elim2}}}{\Longrightarrow} & x_2 \doteq a \cdot x_3,\ a \cdot \alpha x_3 \doteq a \cdot (a \cdot x_4),\ \alpha x_3 \doteq a \cdot x_4,\ \alpha x_4 \doteq a \cdot x_3,\ x_1 \doteq x_4 \\
\overset{R_{\text{bop}}}{\Longrightarrow} & x_2 \doteq a \cdot x_3,\ a \doteq a,\ \alpha x_3 \doteq a \cdot x_4,\ \alpha x_4 \doteq a \cdot x_3,\ x_1 \doteq x_4 \\
\overset{R_{\text{refl}}}{\Longrightarrow} & x_2 \doteq a \cdot x_3,\ \alpha x_3 \doteq a \cdot x_4,\ \alpha x_4 \doteq a \cdot x_3,\ x_1 \doteq x_4
\end{array}
$$

▶ **Proposition 28.** *If $E \Longrightarrow E'$, then E and E' have the same semi-unifiers.*

▶ **Proposition 29.** *If $E, X \doteq s$ is anchored and s is simple, then $E[s/X], X \doteq s$ is anchored.*

▶ **Lemma 30.** *If $E \Longrightarrow E'$ and E is anchored, then E' is anchored.*

Proof. For R_{elim1} and R_{elim2}, this follows immediately from Proposition 29. For the other rules, the claim is trivial. ◀

▶ **Lemma 31.** *There is no infinite reduction sequence $E_1 \Longrightarrow E_2 \Longrightarrow \cdots$ such that E_i is not clashed for all $i \in \{1, 2, \ldots\}$.*

Proof. We assign a triple of natural numbers $(n_1, n_2, n_3 + n_4 + n_5)$ to every system of equations $E = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ where

- n_1 is the number of simple variables in E that are not eliminated,
- n_2 is the number of instance variables in E that are not eliminated,
- n_3 is the sum of the sizes of every term s_i that is not an instance variable for $i \in \{1, 2, \ldots\}$,
- n_4 is the sum of the sizes of every term s_i that is not an atom for $i \in \{1, 2, \ldots\}$ and
- n_5 is the number of equations in E.

With the usual lexicographical ordering, this yields a well-founded ordering on systems of equations. It is easy to show that every rule application respects the ordering and hence an infinite reduction sequence is impossible. ◀

We call a system of equations E *terminal* if there is no system of equations E' with $E \Longrightarrow E'$.

▶ **Proposition 32.** *If E is anchored and terminal, then E is either clashed or in solved form.*

Proof. Let E be anchored, terminal and not clashed. We show that E is in solved form, that is, all equations have the form $X \doteq s$ where X is eliminated and s is simple.

Let $s \doteq t \in E$. Then s is an atom because otherwise R_{bop}, R_{refl}, R_{orient1} or R_{orient2} would be applicable contradicting the fact that E is terminal. Also, t is simple because if it contained an instance variable αx, then E would also contain an equation $\alpha x \doteq u$ because E is anchored. This is a contradiction because R_{elim1} would be applicable. Since s is an atom and t is simple, s must be eliminated because otherwise R_{elim1} or R_{elim2} would be applicable. ◀

Thus we can transform every anchored system of equations E either into solved form or into a clashed system of equations. In the first case, we have computed a principal semi-unifier of E. In the second case, E has no semi-unifier.

11 Conclusion

The paper presents the first unification algorithm for nonnested recursion schemes. Based on a novel coinductive definition of $\mathcal{S}$-equivalence, we establish the existence of principal $\mathcal{S}$-unifiers. Our method for solving $\mathcal{S}$-unification problems works by a reduction to a new decidable semi-unification problem we call AnSUP (anchored semi-unification problem). AnSUP is quite different from other decidable semi-unification problems we know of.

- The uniform fragment [7, 16] only allows for a single substitution variable (or, in the usual representation, a single inequality). In contrast to this, our reduction requires many substitution variables.
- The acyclic fragment [9] and its extension to the R-acyclic fragment [14] disallow cyclic inequalities in the following sense. There must not be a sequence of inequalities $s_1 \dot{\preceq} t_1, \ldots, s_n \dot{\preceq} t_n$ such that t_n and s_1 share a variable and for all $i \in \{1, \ldots, n-1\}$, t_i and s_{i+1} share a variable. In contrast to this, we allow arbitrary cycles and need them in the reduction.
- The left-linear fragment [6] does not allow that a variable occurs twice in the left-hand side s of an inequality $s \dot{\preceq} t$. Adding inequalities of the form $(s, s) \dot{\preceq} (s, t)$ is impossible since this would allow for the same expressive power as unrestricted semi-unification, which is undecidable. So it is impossible to express ordinary unification with the left-linear fragment.
- The quasi-monadic fragment [13] does not allow terms containing two different variables.
- There is a decidable fragment that only allows two variables [11].

To the best of our knowledge, AnSUP is the only fragment that allows for cyclic inequalities, an unrestricted number of variables and subsumes ordinary unification.

Our algorithm for anchored semi-unification needs $\mathcal{O}(n^3)$ steps, since all three components of the triple used in the termination proof (Lemma 31) are linearly bounded by the problem size. However, since our semi-unification rules build on the naive rules for ordinary unification, the term size can grow exponentially in the number of steps. For example, our algorithm performs poorly on the ordinary unification problem $x_1 \doteq x_2 \cdot x_2,\ x_2 \doteq x_3 \cdot x_3, \ldots,\ x_{n-1} \doteq x_n \cdot x_n$. We expect that using the same techniques as for ordinary unification [1], it is possible to design an algorithm for anchored semi-unification with polynomial complexity.

References

1. F. Baader and T. Nipkow. Equational problems. In *Term rewriting and all that*, pages 58–92. Cambridge University Press, 1998.
2. B. Courcelle. A representation of trees by languages II. *Theoretical Computer Science*, 7(1):25–55, 1978.
3. N. Dershowitz, S. Kaplan, and D.A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991.
4. E. Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1(1):31–46, 1985.
5. F. Henglein. Type inference and semi-unification. In *LISP and functional programming*, pages 184–197. ACM, 1988.
6. F. Henglein. Fast left-linear semi-unification. In *Advances in Computing and Information (ICCI'90)*, volume 468 of *Lecture Notes in Computer Science*, pages 82–91. Springer, 1990.
7. D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In *Foundations of Software Technology and Theoretical Computer Science*, pages 435–454. Springer, 1988.
8. A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476. ACM, 1990.
9. A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM (JACM)*, 41(2):368–398, 1994.
10. D.S Lankford and D.R. Musser. A finite termination criterion. *Unpublished draft*, 1978.
11. H. Leiß. Decidability of semi-unification in two variables. Technical report, INF-2-ASE-9-89, Siemens, Munich, 1989.
12. H. Leiß. Polymorphic recursion and semi-unification. In *CSL'89*, pages 211–224. Springer, 1990.
13. H. Leiß and F. Henglein. A decidable case of the semi-unification problem. In *Mathematical Foundations of Computer Science 1991*, volume 520 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 1991.
14. B. Lushman and G.V. Cormack. A larger decidable semiunification problem. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 143–152. ACM, 2007.
15. R.M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.
16. A. Oliart and W. Snyder. Fast algorithms for uniform semi-unification. *Journal of Symbolic Computation*, 37(4):455–484, 2004.
17. P. Pudlák. On a unification problem related to Kreisel's conjecture. *Commentationes Mathematicae Universitatis Carolinae*, 29(3):551–556, 1988.
18. B.K. Rosen. Program equivalence and context-free grammars. *Journal of Computer and System Sciences*, 11(3):358–374, 1975.
19. V. Sabelfeld. The tree equivalence of linear recursion schemes. *Theoretical Computer Science*, 238(1–2):1–29, 2000.
20. D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.
21. G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Automata, languages and programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681. Springer, 1997.
22. C. Stirling. Deciding DPDA equivalence is primitive recursive. In *Automata, languages and programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 774–774. Springer, 2002.

Formalizing Knuth-Bendix Orders and Knuth-Bendix Completion*

Christian Sternagel[1] and René Thiemann[2]

1 **School of Information Science**
Japan Advanced Institute of Science and Technology, Japan
c-sterna@jaist.ac.jp
2 **Institute of Computer Science**
University of Innsbruck, Austria
rene.thiemann@uibk.ac.at

Abstract

We present extensions of our *Isabelle Formalization of Rewriting* that cover two historically related concepts: the Knuth-Bendix order and the Knuth-Bendix completion procedure.

The former, besides being the first development of its kind in a proof assistant, is based on a generalized version of the Knuth-Bendix order. We compare our version to variants from the literature and show all properties required to certify termination proofs of TRSs.

The latter comprises the formalization of important facts that are related to completion, like *Birkhoff's theorem*, the *critical pair theorem*, and a soundness proof of completion, showing that the strict encompassment condition is superfluous for finite runs. As a result, we are able to certify completion proofs.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases certification, completion, confluence, termination

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.287

1 Introduction

In their seminal paper [10], Knuth and Bendix introduced two important concepts: a procedure that allows us to solve certain instances of the word problem – (Knuth-Bendix) completion – as well as a specific order on terms that is useful to orient equations in the aforementioned procedure – the Knuth-Bendix order (or KBO, for short).

Our main contributions around KBO and completion are:

- There are several variants of KBO (e.g., incorporating quasi-precedences, infinite signatures, subterm coefficient functions, and generalized weight functions). In fact, not for all of these variants well-foundedness has been proven. We give the first well-foundedness proof for a variant of KBO that combines infinite signatures, quasi-precedences, and subterm-coefficient functions. Our proof is direct, and we illustrate why we could employ Kruskal's tree theorem only for a less powerful variant of KBO.
- We dropped the strict encompassment condition in the inference rules of completion and present a new proof which shows that the modified rules are still sound for finite runs.

* This research is supported by the Austrian Science Fund (FWF): P22767, J3202.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 287–302

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- All our results have been formalized with the proof assistant Isabelle/HOL [17] as part of our library IsaFoR [25]. That is, we developed the first mechanized proofs of KBO, Birkhoff's theorem [3], and completion.
- We extended our certifier CeTA [25] (whose soundness is formally verified) by the facility to check completion proofs. For accepted proofs, CeTA provides a decision procedure for the word problem.

Note that most termination techniques, besides KBO, that are used by modern termination tools were already formalized in IsaFoR before. Our extensions further increase the percentage of termination proofs (generated by some termination tool) that can be certified and also enables the certification of completion proofs. Moreover, in its current state, IsaFoR covers most of the theorems in the first seven chapters of [1] (only confluence beyond weak orthogonality and decision procedures for special cases like right-ground TRSs are missing), and hence significantly contributes to a full formalization of standard rewriting techniques for first-order rewrite systems.

Both IsaFoR and CeTA are freely available from `http://cl-informatik.uibk.ac.at/software/ceta/`. The results of this paper are available in IsaFoR/CeTA version 2.10.

The paper is organized as follows: In Section 2, we present some preliminaries on term rewriting. Our results on KBO and its formalization are discussed in Section 3. Afterwards, in Section 4, we present our formalization of the critical pair theorem and the algorithm for certifying confluence. These results are required in Section 5, where we prove soundness of completion and illustrate how we certify completion proofs. Our formalization of Birkhoff's theorem is the topic of Section 6, before we conclude and discuss related work in Section 7.

2 Preliminaries

We assume familiarity with rewriting, equational logic, and completion [1].

In the sequel, let R denote an arbitrary binary relation. We say that R is *well-founded* if and only if there is no infinite sequence a, s.t., $a_1\ R\ a_2\ R\ a_3\ R \cdots$.

We call R *almost-full* if and only if all infinite sequences a contain indices $j > i$, s.t., $a_j\ R\ a_i$. The same property is sometimes expressed by saying that all infinite sequences are *good*. (Note that every almost-full relation is reflexive.)

A binary relation $\succsim$ that is reflexive and transitive, is a *quasi-order* (or *preorder*). The *strict part* $\succ$ of $\succsim$ is defined by $x \succ y = x \succsim y \wedge y \not\succsim x$. Two elements x and y are *equivalent* if $x \succsim y$ and $y \succsim x$. A quasi-order whose strict part is well-founded is called a *quasi-precedence*. (The "quasi" part of the name, stems from the fact that different elements may be equivalent.)

A *partial order*, denoted $\succ$, is a binary relation that is transitive and irreflexive. Its reflexive closure is denoted by $\succeq$. If $\succsim$ is a quasi-order, then its strict part $\succ$ is a partial order. In that case, $\succsim\ =\ \succeq$ if and only if $\succsim$ is antisymmetric. (This is in contrast to alternative definitions of partial orders that start from a quasi-order $\succsim$ and additionally require antisymmetry; then we always have $\succsim = \succeq$.)

A *well-partial-order* (*well-quasi-order*) is a partial order $\succ$ (quasi-order $\succsim$), s.t., $\succeq$ ($\succsim$) is almost-full. For every well-partial-order $\succ$ (well-quasi-orders $\succsim$), $\succ$ ($\succ$) is well-founded.

A non-strict order $\succsim$ is *compatible* with a strict order $\succ$ if and only if $\succsim \circ \succ \circ \succsim \subseteq \succ$.

We say that a list $x_1, \ldots, x_m$ is a *superlist* of the list $y_1, \ldots, y_n$ (or equivalently that $y_1, \ldots, y_n$ is embedded in $x_1, \ldots, x_m$), written $x_1, \ldots, x_m \trianglerighteq^* y_1, \ldots, y_n$, if $y_1, \ldots, y_n$ is obtained from $x_1, \ldots, x_m$ by deleting elements (but not changing their order).

A *signature* $\mathcal{F}$ is a set of *function symbols* (denoted by f, g, ... for non-constants and a, b, c, ... for constants). The set of *terms* over a signature $\mathcal{F}$ and a set of variables $\mathcal{V}$ is

denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$. We write $\mathcal{V}_{\mathsf{mul}}(t)$ for the multiset of variables occurring in a term t. Let p be a position in a term t. Then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the result of replacing the subterm of t at position p by the term s. We write $s \trianglerighteq t$ ($s \triangleright t$) if t is a (proper) subterm of t. A *rewrite order* is an irreflexive and transitive order that is closed under contexts and closed under substitutions. For terms s and t from $\mathcal{T}(\mathcal{F},\mathcal{V})$ we call $s \approx t$ an *equation*. An *equational system* (ES) $\mathcal{E}$ is a set of equations.

Sometimes we orient an equation $s \approx t$, write $s \to t$, and call it a *rule*. Then for an ES $\mathcal{E}$ we denote by $\to_{\mathcal{E}}$ the smallest relation that contains $\mathcal{E}$ and is closed under contexts and substitutions. Let $\to$ be a relation. We write $(\to)^{-1}$ (or simply $\leftarrow$) for the inverse of $\to$, $\leftrightarrow$ for $\to \cup \leftarrow$, and $\to^*$ for the reflexive and transitive closure of $\to$.

An ES $\mathcal{E}$ is called a *term rewrite system* (TRS) if all equations are rules. A TRS $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded, and *(locally) confluent* if $({}_{\mathcal{R}}\!\leftarrow \cdot \to_{\mathcal{R}} \subseteq \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow)$ ${}_{\mathcal{R}}^*\!\leftarrow \cdot \to_{\mathcal{R}}^* \subseteq \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow$. A term s is in *($\mathcal{R}$-)normal form* if there is no term t with $s \to_{\mathcal{R}} t$. We write $s{\downarrow_{\mathcal{R}}}$ for an $\mathcal{R}$-normal form of a term s, i.e., some term t such that $s \to_{\mathcal{R}}^* t$ and t is in normal form. Terms s and t are *($\mathcal{R}$-)joinable* if $s \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow t$, and *($\mathcal{E}$-)convertible* if $s \leftrightarrow_{\mathcal{E}}^* t$. A TRS $\mathcal{R}$ and an ES $\mathcal{E}$ are *equivalent* if $\leftrightarrow_{\mathcal{E}}^* = \leftrightarrow_{\mathcal{R}}^*$, i.e., their respective equational theories coincide.

We call (s,t) a *critical pair* of a TRS $\mathcal{R}$ if and only if there are two (not necessarily distinct) variable renamed rules $\ell_i \to r_i$, $i = 1,2$ (without common variables) and a position p in ℓ_1 such that θ is an mgu of $\ell_1|_p$ and ℓ_2, $\ell_1|_p$ is not a variable, $s = r_1\theta$, and $t = \ell_1\theta[r_2\theta]_p$.

An *$\mathcal{F}$-algebra* $\mathcal{A}$ consists of a non-empty set A (the *carrier*) and an *interpretation* $I : \mathcal{F} \to A^* \to A$. For each $\mathcal{F}$-algebra $\mathcal{A} = (A, I)$, and each *variable assignment* $\alpha : \mathcal{V} \to A$, we define the *term evaluation* $[\cdot]_\alpha^{\mathcal{A}} : \mathcal{T}(\mathcal{F},\mathcal{V}) \to A$ as $[x]_\alpha^{\mathcal{A}} = \alpha(x)$ and $[f(t_1,\ldots,t_n)]_\alpha^{\mathcal{A}} = I(f)([t_1]_\alpha^{\mathcal{A}},\ldots,[t_n]_\alpha^{\mathcal{A}})$. We drop $\mathcal{A}$ whenever it is clear from the context.

An $\mathcal{F}$-algebra $\mathcal{A}$ is a *model* of an equation $s \approx t$ if for each variable assignment α the equality $[s]_\alpha = [t]_\alpha$ is holds. If every $\mathcal{F}$-algebra that is a model of all equations in $\mathcal{E}$ also is a model of $s \approx t$, we write $\mathcal{E} \models s \approx t$ and say that $s \approx t$ *follows from* $\mathcal{E}$. The relation $\models$ is called *semantic entailment*.

3 Knuth-Bendix Order

We start by presenting our definition of KBO as it is formalized in IsaFoR in the files `KBO.thy` and `KBO_Impl.thy`. It comes in the form of two mutually recursive functions which define the strict order $\succ_{\mathsf{kbo}}$ and the compatible non-strict order $\succsim_{\mathsf{kbo}}$. The mutual dependency is created by the lexicographic extension of two compatible orders: each of $\succ_{\mathsf{kbo}}^{\mathsf{lex}}$ and $\succsim_{\mathsf{kbo}}^{\mathsf{lex}}$ require both $\succ_{\mathsf{kbo}}$ and $\succsim_{\mathsf{kbo}}$ in their definition.

▶ **Definition 3.1** (Knuth-Bendix Order). Let $\succsim$ be a quasi-precedence, $w_0 \in \mathbb{N} \setminus \{0\}$, and $w : \mathcal{F} \to \mathbb{N}$. Further, let w be *admissible*, i.e., $w(c) \geq w_0$ for all constants c and, if $w(f) = 0$ and f is unary, then f is of largest precedence, i.e., $f \succsim g$ for all g.

The weight function w is lifted to terms as follows.

$$w(x) = w_0$$
$$w(f(t_1,\ldots,t_n)) = w(f) + \sum_{1 \leq i \leq n} w(t_i)$$

We define $s \succ_{\mathsf{kbo}} t$ if and only if $\mathcal{V}_{\mathsf{mul}}(s) \supseteq \mathcal{V}_{\mathsf{mul}}(t)$ and $w(s) \geq w(t)$
- and $w(s) > w(t)$ (weight)
- or $s = f(s_1, \ldots, s_m)$
  - and t is a variable (fun-var)
  - or $t = g(t_1, \ldots, t_n)$
    - and $f \succ g$ (prec)
    - or $f \succsim g$ and $(s_1, \ldots, s_m) \succ^{\mathsf{lex}}_{\mathsf{kbo}} (t_1, \ldots, t_n)$ (lex)

and $s \succsim_{\mathsf{kbo}} t$ if and only if $\mathcal{V}_{\mathsf{mul}}(s) \supseteq \mathcal{V}_{\mathsf{mul}}(t)$ and $w(s) \geq w(t)$
- and $w(s) > w(t)$ (weight)
- or s is a variable
  - and t is a variable (var-var)
  - or t is a constant and $\mathit{least}(t)$ (least)
- or $s = f(s_1, \ldots, s_m)$
  - and t is a variable (fun-var)
  - or $t = g(t_1, \ldots, t_n)$
    - and $f \succ g$ (prec)
    - or $f \succsim g$ and $(s_1, \ldots, s_m) \succsim^{\mathsf{lex}}_{\mathsf{kbo}} (t_1, \ldots, t_n)$ (lex-eq)

Here, *least* is a predicate that checks whether a constant c has weight w_0 and is of least precedence among all constants with weight w_0, i.e., $\forall d.\, w(d) = w_0 \longrightarrow d \succsim c$.

The lexicographic extension, $(s_1, \ldots, s_m) \succ^{\mathsf{lex}}_{\mathsf{kbo}} (t_1, \ldots, t_n)$ is defined by

$$(\exists i \leq \min(m,n).\, s_i \succ_{\mathsf{kbo}} t_i \wedge (\forall j < i.\, s_j \succsim_{\mathsf{kbo}} t_j)) \vee (m > n \wedge (\forall j \leq n.\, s_j \succsim_{\mathsf{kbo}} t_j)), \quad \text{(lex-def)}$$

and its non-strict part $(s_1, \ldots, s_m) \succsim^{\mathsf{lex}}_{\mathsf{kbo}} (t_1, \ldots, t_n)$ is defined similarly, except that $m > n$ is replaced by $m \geq n$.

Note that due to the condition $\mathcal{V}_{\mathsf{mul}}(s) \supseteq \mathcal{V}_{\mathsf{mul}}(t)$, we have that s and t are the same variable in (var-var), and t is a variable occurring in s in (fun-var).

Before we give details of our formalization of KBO, we relate our definition to other definitions of KBO as in [6, 10, 14, 19, 30].

Non-strict Part. In [6, 10, 14, 19, 30] only the strict order $\succ_{\mathsf{kbo}}$, without accompanying $\succsim_{\mathsf{kbo}}$, is defined. In lexicographic comparisons $=$ is used instead of $\succsim_{\mathsf{kbo}}$. However, the usage of $\succsim_{\mathsf{kbo}}$ leads to a more powerful variant of KBO, since $\succsim_{\mathsf{kbo}}$ is reflexive and thus, replacing $=$ by $\succsim_{\mathsf{kbo}}$ can only enlarge $\succ_{\mathsf{kbo}}$. Moreover, our (syntactic) definition of KBO can treat examples that have been used before to illustrate the power of non-syntactic definitions, where $s \succ_{\mathsf{kbo}} t$ for non-ground terms s and t is defined by $s\sigma \succ_{\mathsf{kbo}} t\sigma$ for all ground instances. Using our definition, we can orient the leading example $\mathsf{g}(x, \mathsf{a}, \mathsf{b}) \succ_{\mathsf{kbo}} \mathsf{g}(\mathsf{b}, \mathsf{b}, \mathsf{a})$ of [11] by choosing $w_0 = w(\mathsf{g}) = w(\mathsf{b}) = 1$, $w(\mathsf{a}) = 2$, and a precedence that makes all symbols equivalent.

Quasi-Precedence. In [6, 10, 14] no quasi-precedences are allowed, i.e., $f \succsim g$ is replaced by $f = g$ in (lex) and (lex-eq), which is clearly less powerful.

Lexicographic-Extension. Our definition of lexicographic extension allows comparisons where the decrease is due to a decrease in the lengths of the lists. Of course, this is only relevant for quasi-precedences [19, 30], since otherwise the argument lists always have the same length. Whereas this kind of definition is also used in [19], in [30], a lexicographic decrease is only possible if at some position $i \leq \min(m, n)$ there is a strict decrease.

However, this small change has an important impact: KBO as defined in [30] is neither closed under substitutions nor does it have the subterm property; taking $w_0 = w(\mathsf{a}) = 1$, $w(\mathsf{f}) = 0$ and a precedence where all symbols are equivalent, we have $\mathsf{f}(x) \succ_{\mathsf{kbo}} x$, but not $\mathsf{f}(\mathsf{a}) \succ_{\mathsf{kbo}} \mathsf{a}$, since $(\mathsf{a}) \not\succ^{\mathsf{lex}}_{\mathsf{kbo}} ()$ for the definition of lexicographic extension of [30].

Infinite Signatures. Our definition as well as [19, 30] admit arbitrary, possibly infinite, signatures, whereas in [6, 10, 14], only finite signatures are considered. However, neither [19] nor [30] mention a solution to the problem that Kruskal's tree theorem cannot be easily applied: the result that every rewrite order containing the subterm relation is well-founded, only holds for finite signatures. Hence, a well-foundedness proof for their versions of KBO using infinite signatures is missing.

And indeed, it is questionable whether our definition of KBO on infinite signatures has all the desired properties: the lexicographic extension as defined in (lex-def) does not preserve well-foundedness in general. Consider unbounded arities and take the lexicographic extension of the standard order on natural numbers. Then, we have $(1) >^{\mathsf{lex}} (0, 1) >^{\mathsf{lex}} (0, 0, 1) \ldots$. If we bound arities for lexicographic comparisons, we do not achieve the subterm property: if $w(\mathsf{f}) = 0$ and all symbols have equal precedence, then $\mathsf{f}(\mathsf{g}(\mathsf{a}, \ldots, \mathsf{a})) \succ_{\mathsf{kbo}} \mathsf{g}(\mathsf{a}, \ldots, \mathsf{a})$ can only be shown by proving $(\mathsf{g}(\mathsf{a}, \ldots, \mathsf{a})) \succ^{\mathsf{lex}}_{\mathsf{kbo}} (\mathsf{a}, \ldots, \mathsf{a})$, i.e., for each g of arity n we require a successful comparison of lists of length 1 with lists of length n.

Status Functions. In contrast to [19], we do not integrate a status function which decides how to compare argument lists. Thus, our definition of KBO is incomparable to the one in [19]. The reason for this omission is simple: We are not aware of any termination tool that uses KBO with status. Nevertheless, we do not expect severe difficulties for adding a status function, as we already did this in a formalization of the recursive path order [24].

Ordinal Weights. Another generalization that we do not consider are weight functions over ordinal numbers [14].

To summarize, we are not aware of any proof of well-foundedness for a KBO variant that allows quasi-precedences and infinite signatures. Moreover, the subterm property and closure under substitutions look interesting as they are not satisfied by the definition in [30].

We now state all the desired properties that have been formalized for our version of KBO. Incidentally, we proved all these properties for a variant of KBO that also incorporates the subterm coefficient functions from [14]. That is, for each argument, we can specify how often it should be counted when computing weights and multisets of variables. E.g., for $\Psi(f) = [2, 3]$ we compute the weight function w.r.t. Ψ as $w^\Psi(f(t_1, t_2)) = w(f) + 2w^\Psi(t_1) + 3w^\Psi(t_2)$, and the multiset of variables w.r.t. Ψ as $\mathcal{V}^\Psi_{\mathsf{mul}}(f(t_1, t_2)) = \mathcal{V}^\Psi_{\mathsf{mul}}(t_1) \cup \mathcal{V}^\Psi_{\mathsf{mul}}(t_1) \cup \mathcal{V}^\Psi_{\mathsf{mul}}(t_2) \cup \mathcal{V}^\Psi_{\mathsf{mul}}(t_2) \cup \mathcal{V}^\Psi_{\mathsf{mul}}(t_2)$. Since, for the proofs of the following properties, subterm coefficient functions did not pose any severe challenges, we omit them in the remainder to simplify our presentation.

▶ **Lemma 3.2** (Properties of $\succ_{\mathsf{kbo}}$ and $\succsim_{\mathsf{kbo}}$).

1. *Every term has a weight not less than w_0, i.e., $w(t) \geq w_0$ for all t.*
2. *Strict KBO is irreflexive and non-strict KBO reflexive, i.e., $t \not\succ_{\mathsf{kbo}} t$ and $t \succsim_{\mathsf{kbo}} t$ for all t.*
3. *The non-strict part is an extension of the strict part, i.e., $\succ_{\mathsf{kbo}} \subseteq \succsim_{\mathsf{kbo}}$.*
4. *KBO is closed under contexts, i.e., $s \mathrel{\gtrsim_{\mathsf{kbo}}} t \longrightarrow C[s] \mathrel{\gtrsim_{\mathsf{kbo}}} C[t]$ for all s, t, and C.*
5. *A weak decrease from an argument implies a strict decrease from the overall term, i.e., $s \succsim_{\mathsf{kbo}} t \longrightarrow f(\ldots, s, \ldots) \succ_{\mathsf{kbo}} t$ for all f, s, and t.*
6. *KBO has the* subterm property*, i.e., $\rhd \subseteq \succ_{\mathsf{kbo}}$. (We also have $\trianglerighteq \subseteq \succsim_{\mathsf{kbo}}$.)*

7. *Every term is larger than a least constant, i.e., least*$(c) \longrightarrow t \succsim_{\mathsf{kbo}} c$ *for all* c *and* t.
8. *KBO is closed under substitutions, i.e.,* $s \mathrel{\tilde{\succsim}}_{\mathsf{kbo}} t \longrightarrow s\sigma \mathrel{\tilde{\succsim}}_{\mathsf{kbo}} t\sigma$ *for all* s, t, *and* σ.
9. *All combinations of* $\succ_{\mathsf{kbo}}$ *and* $\succsim_{\mathsf{kbo}}$ *are transitive, i.e.,* $\succsim_{\mathsf{kbo}} \circ \succsim_{\mathsf{kbo}} \subseteq \succsim_{\mathsf{kbo}}$, $\succsim_{\mathsf{kbo}} \circ \succ_{\mathsf{kbo}} \subseteq \succ_{\mathsf{kbo}}$, $\succ_{\mathsf{kbo}} \circ \succsim_{\mathsf{kbo}} \subseteq \succ_{\mathsf{kbo}}$, *and* $\succ_{\mathsf{kbo}} \circ \succ_{\mathsf{kbo}} \subseteq \succ_{\mathsf{kbo}}$

Proof. We only present the proofs of the most interesting properties. All other proofs are provided in IsaFoR, file `KBO.thy`.

5. This is the main auxiliary fact for proving the subterm property. We prove it by induction on t. If t is a variable x, then we have $\mathcal{V}_{\mathsf{mul}}(f(\dots,s,\dots)) \supseteq \mathcal{V}_{\mathsf{mul}}(s) \supseteq \mathcal{V}_{\mathsf{mul}}(x)$, since $s \succsim_{\mathsf{kbo}} t = x$, and $w(f(\dots,s,\dots)) \geq w_0 = w(x)$ follows from **1**. Hence, $f(\dots,s,\dots) \succ_{\mathsf{kbo}} x$ by (fun-var). Otherwise, t is a term of the form $g(t_1,\dots,t_n)$ and $s \succsim_{\mathsf{kbo}} g(t_1,\dots,t_n)$. Using $s \succsim_{\mathsf{kbo}} g(t_1,\dots,t_n)$ we conclude that the conditions on the variables and weights are satisfied when comparing $f(\dots,s,\dots)$ and $g(t_1,\dots,t_n)$, e.g., $w(f(\dots,s,\dots)) \geq w(s) \geq w(g(t_1,\dots,t_n))$. If $w(f(\dots,s,\dots)) > w(g(t_1,\dots,t_n))$ or $f \succ g$ then we are done. Otherwise, $w(f(\dots,s,\dots)) = w(s) = w(g(t_1,\dots,t_n))$ which can only be the case when $w(f) = 0$ and f is unary, i.e., $f(\dots,s,\dots) = f(s)$. By admissibility, we conclude $f \succsim g$ and since $f \not\succ g$ we know $g \succsim f$, and hence, by transitivity of $\succsim$ and admissibility, we know that $g \succsim h$ for all symbols h. Then we consider the following cases:

- **case** $(n = 0)$. Then $f(s) \succ_{\mathsf{kbo}} g = g(t_1,\dots,t_n)$ by (lex), since $(s) \succ^{\mathsf{lex}}_{\mathsf{kbo}} ()$. This is the part of the proof where the length comparisons of lists in $\succ^{\mathsf{lex}}_{\mathsf{kbo}}$ play a crucial role.
- **case** $(n \geq 1)$. We know from $s \succsim_{\mathsf{kbo}} g(t_1,\dots,t_n)$ that s cannot be a variable. So let $s = h(s_1,\dots,s_m)$ for some h and $s_1,\dots,s_m$. Since $w(s) = w(g(t_1,\dots,t_n))$ and $g \succsim h$ we conclude $s_1,\dots,s_m \succsim^{\mathsf{lex}}_{\mathsf{kbo}} t_1,\dots,t_n$. Using $n \geq 1$ and **3** we know that $m \geq 1$ and $s_1 \succsim_{\mathsf{kbo}} t_1$. Using the induction hypothesis, we conclude $s = h(s_1,\dots,s_m) \succ_{\mathsf{kbo}} t_1$ and thus, $(s) \succ^{\mathsf{lex}}_{\mathsf{kbo}} (t_1,\dots,t_n)$ – and in this last comparison one can observe why we must not bound the lengths of the lists in (lex-def). In combination with $f \succsim g$ we conclude $f(s) \succ_{\mathsf{kbo}} g(t_1,\dots,t_n)$ by (lex).

6. From $s \rhd t$, we first obtain a non-empty context such that $s = C[t]$. Then $C[t] \succ_{\mathsf{kbo}} t$ is shown by induction on C, with the help of **2**, **3**, and **5**. Afterwards, the result for $s \trianglerighteq t$ follows by **2** and **3**.

8. We show $(s \succsim_{\mathsf{kbo}} t \longrightarrow s\sigma \succsim_{\mathsf{kbo}} t\sigma) \wedge (s \succ_{\mathsf{kbo}} t \longrightarrow s\sigma \succ_{\mathsf{kbo}} t\sigma)$ to ensure closure under substitutions. The proof works by induction on s followed by a case analysis on t. Most of the proof is straight-forward, just note that we require **7** (if the decrease is due to (least)) and **6** (if the decrease is due to (fun-var)). ◀

It remains to investigate well-foundedness of KBO. For finite signatures, this property follows from Kruskal's tree theorem, as $\succ_{\mathsf{kbo}}$ is a rewrite order which contains the strict subterm relation (and thus is a simplification order for finite signatures). Therefore, the definitions of KBO in [6, 10, 14] are well-founded.

As shown in [16], also for infinite signatures the tree theorem can be employed to show well-foundedness. However, this time the subterm property alone is not enough. Instead we need to regard homeomorphic embedding in the definition of simplification order. Given a relation $\succ$, *homeomorphic embedding* on terms, written $\succ_{\mathsf{he}}$, is defined inductively by the rules

$$\frac{t \in t_1,\dots,t_n}{f(t_1,\dots,t_n) \succ_{\mathsf{he}} t} \qquad \frac{s \succ_{\mathsf{he}} t \quad t \succ_{\mathsf{he}} u}{s \succ_{\mathsf{he}} u} \qquad \frac{s \succ_{\mathsf{he}} t}{C[s] \succ_{\mathsf{he}} C[t]} \qquad \frac{f \succ g \quad s_1,\dots,s_m \trianglerighteq^* t_1,\dots,t_n}{f(s_1,\dots,s_m) \succ_{\mathsf{he}} g(t_1,\dots,t_n)}$$

▶ **Theorem 3.3** (Kruskal's Tree Theorem). *If $\succ$ is a well-partial-order on the signature $\mathcal{F}$, then $\succ_{\mathsf{he}}$ is a well-partial-order on the set of ground terms $\mathcal{T}(\mathcal{F})$.*

Proof. A formalization of Theorem 3.3 is given in [21] and the proofs presented in [20]. ◀

▶ **Definition 3.4** (Simplification Order). A *simplification order* is a rewrite order R, s.t., $\succ_{\mathsf{he}}$ is contained in R, for some well-partial-order $\succ$.

Every simplification order is well-founded. Thus, instead of proving well-foundedness of R directly, it suffices to show that R is a rewrite order that contains $\succ_{\mathsf{he}}$ for some well-partial-order $\succ$.

We were able to show (and formalize in Isabelle/HOL) the following theorem.

▶ **Theorem 3.5.** *If $\succ$ is a well-partial-order, then $\succ_{\mathsf{kbo}}$ is a simplification order (and thus well-founded).*

Proof. The proof of [16, Theorem 7.10] works also for our definition. ◀

For definitions of KBO that are not based on a quasi-precedence (e.g., [16]), the above theorem is enough to show that any KBO $\succ_{\mathsf{kbo}}$ over a well-founded partial order $\succ$ is a simplification order. This can be seen as follows: every well-founded partial order $\succ$ can be extended to a total well-order $\succ'$ (i.e., a well-founded partial order that satisfies $\forall x\, y.\, x = y \vee x \succ' y \vee y \succ' x$); moreover, every total well-order is a well-partial-order.

▶ **Lemma 3.6.** *Every total well-order is a well-partial-order.*

Proof. Let $\succ$ be a total well-order. Assume that it is not a well-partial-order (i.e., not almost-full). Then there is an infinite sequence a such that $a_j \not\succ a_i$ for all $j > i$. By totality, we have $a_i \succ a_j$ for all $j > i$ and thus $a_1 \succ a_2 \succ a_3 \succ \cdots$, contradicting well-foundedness. ◀

Furthermore, KBO is monotone w.r.t. the given precedence (i.e., $\succ \subseteq \succ'$ implies $\succ_{\mathsf{kbo}} \subseteq \succ'_{\mathsf{kbo}}$); thus, we can apply Theorem 3.5 to obtain well-foundedness of $\succ'_{\mathsf{kbo}}$ and then, by $\succ_{\mathsf{kbo}} \subseteq \succ'_{\mathsf{kbo}}$, conclude well-foundedness of $\succ_{\mathsf{kbo}}$.

Unfortunately, the same procedure is not applicable with our definition of KBO, since it does not seem to be monotone w.r.t. the given quasi-precedence, unless $\succsim$ is antisymmetric and thus $\succsim = \succeq$ (and we are back at not having a quasi-precedence).

Unfortunately, that means that Theorem 3.5 is not relevant for termination tools, since those typically provide some quasi-precedence, but do not care, whether its strict part is a well-partial-order.

Besides the problems with quasi-precedences, further note that the proof of Theorem 3.5 does not apply in the presence of subterm coefficient functions. Therefore, we also formalized a direct well-foundedness proof which has some similarities to [16] and integrates all extensions: arbitrary signatures, quasi-precedences, and subterm coefficient functions (again, the last extension is only visible in IsaFoR).

▶ **Theorem 3.7** (Well-Foundedness). *KBO is well-founded, i.e.,* $\mathrm{SN}(\succ_{\mathsf{kbo}})$.

Proof. Let *strong normalization of a term t*, written $\mathrm{SN}(t)$, be the property that there is no infinite sequence $t \succ_{\mathsf{kbo}} t_1 \succ_{\mathsf{kbo}} t_2 \succ_{\mathsf{kbo}} \ldots$. Then it suffixes to show that for all terms s, we have $\mathrm{SN}(s)$. Let s be an arbitrary but fixed term. We perform four inductions, labeled (I), (II), (III), and (IV).

The outermost induction (I) is on s. The variable case is easy as variables are normal forms w.r.t. $\succ_{\mathsf{kbo}}$. For the non-variable case we have to show $\mathrm{SN}(s_1, \ldots, s_m) \longrightarrow$

$\mathrm{SN}(f(s_1,\ldots,s_m))$ where $\mathrm{SN}(s_1,\ldots,s_m)$ abbreviates $\forall s \in s_1,\ldots,s_m.\,\mathrm{SN}(s)$. This property is proven by induction (II) on f and $s_1,\ldots,s_m$ where the induction relation compares the pairs $(w(f(s_1,\ldots,s_m)),f)$ lexicographically, using the standard order $>$ on natural numbers for the first component, and $\succ$ to compare the function symbols in the second component.

In principle, we would like to add a third component to the lexicographic comparisons, where we compare the argument lists $s_1,\ldots,s_m$ by $\succ_{\mathsf{kbo}}^{\mathsf{lex}}$. But $\succ_{\mathsf{kbo}}^{\mathsf{lex}}$ is not necessarily well-founded, as we did not bound the lengths of these lists. We bound the lengths in a further induction (III). To be more precise, once, we have to prove the property of induction (II) for some given f and $s_1,\ldots,s_m$, we define $\succ_{\mathsf{kbo}}^{\mathsf{lexb}}$ as lexicographic extension of $\succ_{\mathsf{kbo}}$ where the lengths of all lists are bounded by $w(f(s_1,\ldots,s_m))$ and where it is assumed that all elements in the lists are already strongly normalizing. Then indeed, $\succ_{\mathsf{kbo}}^{\mathsf{lexb}}$ is strongly normalizing, so we can prove the following property by induction (III) on $t_1,\ldots,t_n$ for all g w.r.t. the induction relation $\succ_{\mathsf{kbo}}^{\mathsf{lexb}}$.

$$f \succsim g \longrightarrow w(f(s_1,\ldots,s_m)) \geq w(g(t_1,\ldots,t_n)) \longrightarrow \mathrm{SN}(t_1,\ldots,t_n) \longrightarrow \mathrm{SN}(g(t_1,\ldots,t_n)) \quad (\star)$$

Clearly, once this property is proven, we can finish induction (II) by choosing $g = f$ and $t_1,\ldots,t_n = s_1,\ldots,s_m$.

To prove $(\star)$, we assume the preconditions of $(\star)$ for some g and $t_1,\ldots,t_n$ and have to show $\mathrm{SN}(g(t_1,\ldots,t_n))$. To this end, we prove that all successors of $g(t_1,\ldots,t_n)$ are strongly normalizing, i.e., for all u: $g(t_1,\ldots,t_n) \succ_{\mathsf{kbo}} u \longrightarrow \mathrm{SN}(u)$. This property we show by induction (IV) which is a structural induction on u.

There is nothing to show if u is a variable, so let $u = h(u_1,\ldots,u_k)$. Since $g(t_1,\ldots,t_n) \succ_{\mathsf{kbo}} h(u_1,\ldots,u_k)$ we also know that $g(t_1,\ldots,t_n)$ is larger than every element of $u_1,\ldots,u_k$ using the subterm property and transitivity. Hence, using induction hypothesis (IV) we know $\mathrm{SN}(u_1,\ldots,u_k)$.

Now, if $(w(f(s_1,\ldots,s_m)),f)$ is larger than $(w(h(u_1,\ldots,u_k),h)$ w.r.t. induction relation (II), then we are done using the induction hypothesis (II). Otherwise, in combination with the preconditions of $(\star)$ and $g(t_1,\ldots,t_n) \succ_{\mathsf{kbo}} h(u_1,\ldots,u_k)$ we conclude $w(f(s_1,\ldots,s_m)) \geq w(h(u_1,\ldots,u_k))$, $f \succsim h$, and $(t_1,\ldots,t_n) \succ_{\mathsf{kbo}}^{\mathsf{lex}} (u_1,\ldots,u_k)$. Since both $w(h(u_1,\ldots,u_k))$ and $w(g(t_1,\ldots,t_n))$ are smaller than $w(f(s_1,\ldots,s_m))$ in particular this shows, that the lengths of $u_1,\ldots,u_k$ and $t_1,\ldots,t_n$ are smaller than $w(f(s_1,\ldots,s_m))$. As additionally both $\mathrm{SN}(u_1,\ldots,u_k)$ and $\mathrm{SN}(t_1,\ldots,t_n)$ we can derive $(t_1,\ldots,t_n) \succ_{\mathsf{kbo}}^{\mathsf{lexb}} (u_1,\ldots,u_k)$ from $(t_1,\ldots,t_n) \succ_{\mathsf{kbo}}^{\mathsf{lex}} (u_1,\ldots,u_k)$. But then induction hypothesis (III) can be applied to derive the desired property $\mathrm{SN}(h(u_1,\ldots,u_k))$. ◀

4 Confluence

In this section we first discuss challenges in the formalization of the critical pair theorem (which is essential to prove soundness of completion), and afterwards present our algorithm to actually certify confluence proofs (which is reused later to certify completion proofs). The corresponding formalizations are provided in files `Critical_Pairs.thy` and `Critical_Pairs_Impl.thy`.

4.1 Critical Pair Theorem

Recall that in the context of completion, we are interested in terminating and confluent TRSs. Using our formalization of KBO (and other termination criteria that are available in IsaFoR), we can already certify termination proofs. Hence, we only require a criterion

to check for confluence. By Newman's lemma (which is trivially formalized), it suffices to analyze local confluence. Here, the key technique is the critical pair theorem of Huet [9] – making a result by Knuth and Bendix [10] explicit. It states that $\mathcal{R}$ is locally confluent if and only if all critical pairs of $\mathcal{R}$ are joinable.

By definition, every critical pair (s, t), gives rise to a peak

$$s = r_1\theta \;{}_{\mathcal{R}}\!\leftarrow \ell_1\theta = \ell_1\theta[\ell_1|_p\theta]_p = \ell_1\theta[\ell_2\theta]_p \rightarrow_{\mathcal{R}} \ell_1\theta[r_2\theta]_p = t.$$

Hence, for local confluence all critical pairs must be joinable. Huet, Knuth, and Bendix have shown that for local confluence it is indeed sufficient to show joinability of critical pairs: whenever $t_1 \;{}_{\mathcal{R}}\!\leftarrow s \rightarrow_{\mathcal{R}} t_2$ where t_1 and t_2 are not joinable, then there are C, σ, t_1', and t_2' such that $t_1 = C[t_1'\sigma]$, $t_2 = C[t_2'\sigma]$, and (t_1', t_2') or (t_2', t_1') is a critical pair of $\mathcal{R}$.

In order to formalize the notion of critical pairs in Isabelle, we need a unification algorithm. To this end, we have formalized the algorithm of [1, Figure 4.5]. In contrast to [1] we directly proved all properties (termination, soundness, and completeness) for this concrete algorithm (as opposed to the abstract algorithm of [1, Chapter 4.6], which transforms unification problems). As a result, the termination argument is a bit easier (it does not rely upon the notion of solved variable), whereas the soundness proof becomes a bit harder (as we had to prove that every result of the unification algorithm is in solved form), cf. `Substitution.thy`.

Having unification, a problem when formalizing the definition of critical pairs occurred. Notice that in IsaFoR, terms are polymorphic in the set of function symbols (all elements of type α) and the set of variables $\mathcal{V}$ (all elements of type β). Hence, to prove the critical pair theorem in a generic way we cannot assume anything on the set of variables. In particular, $\mathcal{V}$ might be finite and thus, we might not even have enough variables for building the critical pairs (as their definition requires variable renamed rules) and thus cannot even formulate the critical pair theorem for arbitrary $\mathcal{V}$.[1]

Since the definition of critical pairs should be executable, we actually do not only want that there are enough variables for the renaming, but we want some executable algorithm which actually performs such a renaming. Therefore, we did not formalize the critical pair theorem for arbitrary infinite sets $\mathcal{V}$, but for strings. For this type of variables it was easy to define an efficient and executable renaming algorithm: it just prefixes all variables by x in the one rule, and by y in the other rule.

▶ **Theorem 4.1.** *A TRS $\mathcal{R}$ over $\mathcal{T}(\mathcal{F}, String)$ is locally confluent if and only if all critical pairs of $\mathcal{R}$ are joinable.*

Note that the theorem does not require any variable-condition for $\mathcal{R}$. Hence, $\mathcal{R}$ may, e.g., contain left-hand sides which are variables or free variables in right-hand sides.

Further note that there are prior formalizations of the critical pair theorem in other theorem provers. The first one is described in [18] using ACL2, and another one is presented in [7] using PVS. Our own formalization is similar to the one in [7], as both are based on the higher-order structure of Huet's proof. However, there are some differences to [7], e.g., in the definition of critical pairs. Whereas in [7] arbitrary renaming substitutions and most general unifiers are allowed for building critical pairs, our definition uses specific renaming and unification algorithms. As a consequence, for a finite TRS we only get finitely many critical pairs, whereas in [7] one usually has to consider infinitely many critical pairs (which arise from using different variable names in the substitutions).

[1] Also other well-known results are problematic if there are not enough variables. For example, during our formalization we figured out that infinitely many variables are also essential for Toyama's modularity result on confluence [27].

4.2 Certifying Joinability

To actually certify local confluence proofs, we have to ensure that all critical pairs are joinable. There are two possibilities.

- The certificate provides the joining rewrite sequences for each critical pair.
- The certifier itself computes joining sequences.

The first possibility has the advantage that it is easy to formalize. However, it will make the certificates bulky. Even more important, this approach cannot be used to prove non-confluence, since it does not allow to certify that a critical pair is not joinable.

Both disadvantages are resolved in the second approach, where due to termination, for each critical pair (s, t) we just have to check $s{\downarrow_{\mathcal{R}}} = t{\downarrow_{\mathcal{R}}}$. Hence, in IsaFoR we integrated an automatic check which computes normal forms.

This sounds like a trivial task, where we just define:

$$\textit{compute-NF}\ \mathcal{R}\ s \equiv \mathsf{if}\ \{t \mid s \to_{\mathcal{R}} t\} = \varnothing\ \mathsf{then}\ t\ \mathsf{else}\ \textit{compute-NF}\ \mathcal{R}\ (\mathsf{SOME}\ t.\, s \to_{\mathcal{R}} t)$$

The problem is that *compute-NF* is not a total function (since the parameter $\mathcal{R}$ might be a nonterminating TRS). Hence, the function package of Isabelle [13] accepts *compute-NF* only as partial function, where the defining equation of *compute-NF* is guarded by a condition (if the input belongs to the domain, i.e., $\mathcal{R}$ is terminating, then the equation is valid). However, conditional equations prevent to use the code generation facility of Isabelle [8]. Thus, the above definition is not suitable for our setting.

The solution is to use the partial function command of [12]. It allows to define partial functions which can be code generated, if the functions have a specific shape. In our case, we just have to wrap the result of *compute-NF* in an option-type (where *None* represents nontermination) and everything works fine. For the generated code this means that a call to *compute-NF* may diverge in general. However, we invoke *compute-NF* only if termination of $\mathcal{R}$ has already been ensured.

5 Knuth-Bendix Completion

As in the previous section on confluence, we start with the formalization of the general soundness theorem of completion, and afterwards discuss the certification algorithm for checking completion proofs. The formalizations are available in files `Completion.thy` and `Check_Completion_Proof.thy`.

5.1 Soundness of Completion

Having formalized the critical pair theorem, we proceed to also formalize soundness of Knuth-Bendix completion. Since in the end, we are only able to certify finite completion runs (as the input cannot be infinite), we only formalized soundness for finite runs. To this end, we used the completion rules from [1] as illustrated in Figure 1. However, the restriction to finite runs allowed us to drop the requirement of strict encompassment in the collapse-rule, cf. Theorem 5.2.

We write $(\mathcal{E}_i, \mathcal{R}_i) \rightsquigarrow (\mathcal{E}_{i+1}, \mathcal{R}_{i+1})$ for the application of an arbitrary inference rule to the pair $(\mathcal{E}_i, \mathcal{R}_i)$ resulting in $(\mathcal{E}_{i+1}, \mathcal{R}_{i+1})$.

▶ **Definition 5.1.** A completion *run* for $\mathcal{E}$ is a finite sequence $(\mathcal{E}_0, \mathcal{R}_0) \rightsquigarrow^n (\mathcal{E}_n, \mathcal{R}_n)$ of rule applications, where $\mathcal{E}_0 = \mathcal{E}$ and $\mathcal{R}_0 = \varnothing$. A run is *successful* if $\mathcal{E}_n = \varnothing$ and every critical pair of $\mathcal{R}_n$ is contained in $\bigcup_{i \le n} \mathcal{E}_i$ or is joinable by $\mathcal{R}_n$.

$$\frac{(\mathcal{E},\mathcal{R})}{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}\ (\mathsf{deduce})\quad \text{if } s\ {}_{\mathcal{R}}\!\leftarrow u\rightarrow_{\mathcal{R}} t \qquad \frac{(\mathcal{E}\cup\{s\approx s\},\mathcal{R})}{(\mathcal{E},\mathcal{R})}\ (\mathsf{delete})$$

$$\frac{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}{(\mathcal{E},\mathcal{R}\cup\{s\to t\})}\ (\mathsf{orient}_{\mathsf{l}})\quad \text{if } s\succ t \qquad \frac{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}{(\mathcal{E},\mathcal{R}\cup\{t\to s\})}\ (\mathsf{orient}_{\mathsf{r}})\quad \text{if } t\succ s$$

$$\frac{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}{(\mathcal{E}\cup\{u\approx t\},\mathcal{R})}\ (\mathsf{simplify}_{\mathsf{l}})\quad \text{if } s\rightarrow_{\mathcal{R}} u \qquad \frac{(\mathcal{E}\cup\{s\approx t\},\mathcal{R})}{(\mathcal{E}\cup\{s\approx u\},\mathcal{R})}\ (\mathsf{simplify}_{\mathsf{r}})\ \text{if } t\rightarrow_{\mathcal{R}} u$$

$$\frac{(\mathcal{E},\mathcal{R}\cup\{s\to t\})}{(\mathcal{E},\mathcal{R}\cup\{s\to u\})}\ (\mathsf{compose})\ \text{if } t\rightarrow_{\mathcal{R}} u \qquad \frac{(\mathcal{E},\mathcal{R}\cup\{s\to t\})}{(\mathcal{E}\cup\{u\approx t\},\mathcal{R})}\ (\mathsf{collapse})\ \text{if } s\rightarrow_{\mathcal{R}} u$$

Figure 1 The inference rules of completion.

If $(\mathcal{E},\varnothing)\rightsquigarrow^n(\varnothing,\mathcal{R}_n)$ is a successful run for $\mathcal{E}$, then $\mathcal{R}_n$ is confluent, terminating, and equivalent to $\mathcal{E}$ (cf. Theorem 5.2). Note that the requirement on critical pairs for a successful run can be replaced by one of the two weaker criteria: either local confluence of $\mathcal{R}_n$ (all critical pairs are joinable) or generation of all critical pairs of $\mathcal{R}_n$ during the run (all critical pairs are contained in $\bigcup_{i\le n}\mathcal{E}_i$).

We have formally proven soundness of completion in IsaFoR.

▶ **Theorem 5.2** (Soundness of Completion). *If $(\mathcal{E},\varnothing)\rightsquigarrow^n(\varnothing,\mathcal{R})$ is a successful run of completion then $\mathcal{R}$ is confluent, terminating, and equivalent to $\mathcal{E}$.*

Proof. Let $(\mathcal{E},\varnothing)\rightsquigarrow^n(\varnothing,\mathcal{R})$ be a successful run. Then there are $\mathcal{E}_i$ and $\mathcal{R}_i$ with $(\mathcal{E}_i,\mathcal{R}_i)\rightsquigarrow(\mathcal{E}_{i+1},\mathcal{R}_{i+1})$ for all $0\le i<n$, $\mathcal{E}_0=\mathcal{E}$, $\mathcal{R}_0=\varnothing$, $\mathcal{E}_n=\varnothing$, and $\mathcal{R}_n=\mathcal{R}$. In total, we have to prove that $\mathcal{R}$ and $\mathcal{E}$ are equivalent, that $\mathcal{R}$ is terminating, and that $\mathcal{R}$ is confluent. Here, for equivalence and termination we just formalized the proofs which are provided in [1] since they do not depend on the encompassment condition in the original collapse rule.

The real interesting part is to prove confluence of $\mathcal{R}$, where we use proof orders [2], which are also utilized in [1]. More specifically, we formalized the proof using the same structure as in the proof of [1, Lemma 7.3.4] and just list the most important differences.

In our case, the persistent rules $\mathcal{R}_\omega$ are $\mathcal{R}_n$, the set $\mathcal{R}_\infty$ is $\bigcup_{i\le n}\mathcal{R}_i$, the persistent equations are $\mathcal{E}_\omega=\mathcal{E}_n=\varnothing$, and $\mathcal{E}_\infty=\bigcup_{i\le n}\mathcal{E}_i$.

The cost of a proof step (s_{i-1},s_i) is just a pair and not a triple as in [1], where we keep the first component of the original triple, but drop the part that is concerned with strict encompassment. Instead, we use another well-founded order on rules. To this end, we define the latest occurrence function as $o(\cdot)$ where $o(\ell\to r)=\max\{i\mid i\le n,\ell\to r\in\mathcal{R}_i\}$. Then we define the cost $c(\cdot)$ of a proof step as follows: if $s_{i-1}\leftrightarrow_{\mathcal{E}_\infty}s_i$ then $c(s_{i-1},s_i)=(\{s_{i-1},s_i\},-)$ where the first component is a multiset of terms and the second component is irrelevant; if $s_{i-1}\ {}^{\ell\to r}\!\!\rightarrow_{\mathcal{R}_\infty}s_i$ then $c(s_{i-1},s_i)=(\{s_{i-1}\},n-o(\ell\to r))$; if $s_{i-1}\ {}^{\ell\to r}\!\!\leftarrow_{\mathcal{R}_\infty}s_i$ then $c(s_{i-1},s_i)=(\{s_i\},n-o(\ell\to r))$.

As in the original proof, these pairs are compared lexicographically, where the first component is compared via the multiset extension of the reduction order $\succ$, and the second using the standard greater-than order on natural numbers. Hence, the order on the cost of proof steps is well-founded. Finally, as in the original proof, the cost of a conversion proof $s\leftrightarrow^*_{\mathcal{E}_\infty\cup\mathcal{R}_\infty}t$ is the multiset of costs of each single proof step, and these conversion costs are compared via the multiset extension of the cost of a single proof step. This defines the relation $\succ_{\mathcal{C}}$ on conversion proofs which is well-founded.

In [1, Lemma 7.3.4] it is stated that whenever there is a conversion $s \leftrightarrow^*_{\mathcal{E}_\infty \cup \mathcal{R}_\infty} t$ which is not a rewrite proof of $\mathcal{R}_\omega$, i.e., a conversion of the form $s \to^*_{\mathcal{R}_\omega} \cdot {}^*_{\mathcal{R}_\omega}\!\leftarrow t$, then there is another conversion between s and t with less cost. This is performed via a large case analysis (cases 1.1, 1.2, 1.3, 2.1, 2.2, 3.1, and 3.2). This proof can almost completely be formalized in our setting with the new collapse-rule and the different cost function. The reason is that in cases 1.1, 1.2, 1.3, 3.1, and 3.3, collapse does not occur and moreover, there is a decrease of cost due to the first component in the lexicographic combination – and as we have the identical first component, we obtain a decrease of cost by the same reasoning.

Hence, it remains to investigate cases 2.1 and 2.2 where the precondition of both cases is that $s_{i-1} \to_{\mathcal{R}_\infty} s_i$ at position p using a rule $s \to t \in \mathcal{R}_\infty - \mathcal{R}_\omega$ and substitution σ. Let $k = o(s \to t)$. Thus, $c(s_{i-1}, s_i) = (\{s_{i-1}\}, n - k)$. By definition of o and the fact that $s \to t \notin \mathcal{R}_\omega = \mathcal{R}_n$, we conclude $k < n$. Hence, in the step from $(\mathcal{E}_k, \mathcal{R}_k)$ to $(\mathcal{E}_{k+1}, \mathcal{R}_{k+1})$ the rule $s \to t$ is removed from $\mathcal{R}_k$, so let $\mathcal{R}_k = \mathcal{R}' \cup \{s \to t\}$.

Now we consider both cases how $s \to t$ has been removed in the k-th step.

If we used compose (case 2.1) then $s \to t$ has been replaced by $s \to u$ since $t \overset{\ell \to r}{\to}_{\mathcal{R}'} u$ for some $\ell \to r \in \mathcal{R}'$. As $s \to u \in \mathcal{R}_{k+1}$ we know that $o(s \to u) > k$ and thus, $n - k > n - o(s \to u)$. We follow the original proof by replacing the one proof step $s_{i-1} \to_{\mathcal{R}_\infty} s_i$ by the two proof steps $s_{i-1} = s_{i-1}[s\sigma]_p \overset{s \to u}{\to}_{\mathcal{R}_\infty} s_{i-1}[u\sigma]_p \overset{\ell \to r}{\leftarrow}_{\mathcal{R}'} s_{i-1}[t\sigma]_p = s_i$. Since the cost of the original proof step $c(s_{i-1}, s_i) = (\{s_{i-1}\}, n - k)$ is strictly larger than the cost of both new proof steps – $c(s_{i-1}, s_{i-1}[u\sigma]_p) = (\{s_{i-1}\}, n - o(s \to u))$ yields a decrease in the second component and $c(s_{i-1}[u\sigma]_p, s_i) = (\{s_i\}, n - o(\ell \to r))$ yields a decrease in the first component – we have constructed a conversion with smaller cost.

Similarly, if we used collapse (case 2.2) then $s \to t$ has been replaced by $u \approx t$ since $s \overset{\ell \to r}{\to}_{\mathcal{R}'} u$ for some $\ell \to r \in \mathcal{R}'$. As $\ell \to r \in \mathcal{R}' = \mathcal{R}_{k+1}$ we know that $o(\ell \to r) > k$ and thus, $n - k > n - o(\ell \to r)$. We follow the original proof by replacing the one proof step $s_{i-1} \to_{\mathcal{R}_\infty} s_i$ by the two proof steps $s_{i-1} = s_{i-1}[s\sigma]_p \overset{\ell \to r}{\to}_{\mathcal{R}_\infty} s_{i-1}[u\sigma]_p \overset{u \approx t}{\leftrightarrow}_{\mathcal{E}_\infty} s_{i-1}[t\sigma]_p = s_i$. Since the cost of the original proof step $c(s_{i-1}, s_i) = (\{s_{i-1}\}, n-k)$ is strictly larger than the cost of both new proof steps – $c(s_{i-1}, s_{i-1}[u\sigma]_p) = (\{s_{i-1}\}, n - o(\ell \to r))$ yields a decrease in the second component and $c(s_{i-1}[u\sigma]_p, s_i) = (\{s_{i-1}[u\sigma], s_i\}, -)$ yields a decrease in the first component – we have constructed a conversion with smaller cost.

In total, also for our completion rules, we can prove the major lemma that whenever there is a conversion $s \leftrightarrow^*_{\mathcal{E}_\infty \cup \mathcal{R}_\infty} t$ which is not a rewrite proof of $\mathcal{R}_\omega$, then there is another conversion between s and t with less cost w.r.t. $\succ_{\mathcal{C}}$. Using this result it is straightforward to show confluence of $\mathcal{R}_n = \mathcal{R}_\omega$.

Since $\mathcal{R}_n$ is terminating it suffices to prove local confluence. By the critical pair theorem we only have to prove joinability of all critical pairs. So, let (s, t) be some critical pair of $\mathcal{R}_n$. If it is not joinable, then by the definition of a successful run we know that $(s, t) \in \mathcal{E}_\infty$ and thus, $s \leftrightarrow^*_{\mathcal{E}_\infty \cup \mathcal{R}_\infty} t$. Since $\succ_{\mathcal{C}}$ is well-founded and by using the major lemma we know that there also is a conversion of s and t which is a rewrite proof of $\mathcal{R}_n$. But this is the same as demanding that s and t are joinable via $\mathcal{R}_n$. ◀

5.2 Certification of Completion Proofs

For checking completion proofs, although Theorem 5.2 has been formalized, it is not checked whether the completion rules are applied correctly. This allows us to also certify proofs from completion tools which are based on other inference rules than those in Figure 1. Instead it is just verified whether the result of the completion procedure is a confluent and terminating TRS that is equivalent to the initial set of equations. How to ensure confluence was already

described in Section 4.2, and for termination we use the existing machinery of IsaFoR [26].

It remains to check that $\mathcal{R}$ is equivalent to $\mathcal{E}$ which is done by showing $\ell \leftrightarrow^*_{\mathcal{E}} r$ for all $\ell \to r \in \mathcal{R}$ and $s \leftrightarrow^*_{\mathcal{R}} t$ for all $s \approx t \in \mathcal{E}$.

For the latter, since we already know that $\mathcal{R}$ is confluent and terminating, we just compare the normal forms of s and t in the same way as we check joinability of critical pairs in Section 4.2.

For the former, we demand a step-wise conversion from ℓ to r as input. Such a conversion can be obtained by storing additional information during the completion run, e.g., a history of all applied inference rules.

Given some successful completion proof from $\mathcal{E}$ to $\mathcal{R}$, we can of course also easily implement the decision procedure for $s \leftrightarrow^*_{\mathcal{E}} t$. In combination with a formalization of Birkhoff's theorem in the upcoming section, we obtain a formalized decision procedure for $\mathcal{E} \models s \approx t$. To be more precise, in the file `Check_Equational_Proof.thy` we provide two means to prove and disprove $\mathcal{E} \models s \approx t$, namely by checking a completion proof and afterwards check whether the normal forms of s and t coincide or not. For ensuring $\mathcal{E} \models s \approx t$ one can alternatively provide a derivation $s \leftrightarrow^*_{\mathcal{E}} t$, or a proof tree using the inference rules of equational logic.

6 Birkhoff's Theorem

Birkhoff's theorem [3] states that semantic entailment and convertibility are equivalent: $\mathcal{E} \models s \approx t$ if and only if $s \leftrightarrow^*_{\mathcal{E}} t$. To show this theorem we had to formalize semantic entailment. Already at this point we encountered the first problem. Whereas it was easy to formalize term evaluation and whether an $\mathcal{F}$-algebra (A, I) is a model of an equation (written $I \models s \approx t$), a problem occurred in the definition of semantic entailment.

Before describing the problem, we show how $\mathcal{F}$-algebras are formalized in IsaFoR, within the file `Equational_Reasoning.thy`. As shown in Section 2 we consider $\mathcal{F}$-algebras consisting of a non-empty carrier A and an interpretation I. In Isabelle, a (non-dependently) typed higher-order logic, we represent such interpretations by functions of type $\alpha \Rightarrow \gamma\ \mathit{list} \Rightarrow \gamma$ (where lower-case Greek letters denote type variables and $\gamma\ \mathit{list}$ is the type of finite lists having elements of type γ) which corresponds to $\mathcal{F} \to A^* \to A$ from Section 2. More specifically, the carrier A as well as the signature $\mathcal{F}$ are implicit in the type. In Isabelle it is not possible to quantify over types. Hence, we cannot define semantic entailment by

$$\forall \gamma.\, \forall I :: \alpha \Rightarrow \gamma\ \mathit{list} \Rightarrow \gamma.\, ((\forall s' \approx t' \in \mathcal{E}.\, I \models s' \approx t') \longrightarrow I \models s \approx t) \tag{1}$$

To solve this problem we just omit the outer quantifier and define $\mathcal{E} \models s \approx t$ by

$$\forall I :: \alpha \Rightarrow \gamma\ \mathit{list} \Rightarrow \gamma.\, ((\forall s' \approx t' \in \mathcal{E}.\, I \models s' \approx t') \longrightarrow I \models s \approx t) \tag{2}$$

where γ, the type of carrier elements, is a type variable. To make this dependence more prominent, we write $\mathcal{E} \models_\gamma s \approx t$ in the following.

Using (2) we proved one direction of Birkhoff's theorem – whenever $s \leftrightarrow^*_{\mathcal{E}} t$ then $\mathcal{E} \models_\gamma s \approx t$ – via induction over the length of the conversion $s \leftrightarrow^*_{\mathcal{E}} t$. Since in this theorem γ is arbitrary, we have indeed formalized that provability implies semantic entailment as defined in Section 2.

For the other direction, we shortly recall the main ideas of the proof of Birkhoff's theorem as presented in [1]. We construct an $\mathcal{F}$-algebra where the carrier is the set of equivalence classes of $\leftrightarrow^*_{\mathcal{E}}$, i.e., $A = \mathcal{T}(\mathcal{F}, \mathcal{V})/{\leftrightarrow^*_{\mathcal{E}}}$ and every symbol is interpreted by itself, i.e., $I(f)([t_1]_{\leftrightarrow^*_{\mathcal{E}}}, \ldots, [t_n]_{\leftrightarrow^*_{\mathcal{E}}}) = [f(t_1, \ldots, t_n)]_{\leftrightarrow^*_{\mathcal{E}}}$. Note that I is well-defined, since $\leftrightarrow^*_{\mathcal{E}}$ is a congruence. It can be shown that (A, I) is a model of all equations in $\mathcal{E}$. Hence, whenever $\mathcal{E} \models s \approx t$, then $[s]_{\leftrightarrow^*_{\mathcal{E}}} = [t]_{\leftrightarrow^*_{\mathcal{E}}}$ and thus, $s \leftrightarrow^*_{\mathcal{E}} t$.

The problem in the formalization of this proof is that the carrier $A = \mathcal{T}(\mathcal{F}, \mathcal{V})/{\leftrightarrow_{\mathcal{E}}^*}$ is not expressible as type in Isabelle. The reason is that quotient types can only be built statically, and may not depend on a dynamic input like an arbitrary set of equations $\mathcal{E}$. To this end, in the formalization we replaced (2) by

$$\forall A :: \gamma\ set.\, \forall I.\, well\text{-}defined_A(I) \wedge (\forall s' \approx t' \in \mathcal{E}.\, I \models s' \approx t') \longrightarrow I \models s \approx t \tag{3}$$

where the carrier A is not a type, but an arbitrary set having elements from the parametric type γ. Here, the predicate $well\text{-}defined_A(I)$ checks whether the interpretation I is well-defined w.r.t. the carrier A, i.e., for every f it must be ensured that $I(f)(a_1, \ldots, a_n) \in A$ whenever each $a_i \in A$. In (1) and (2) this was automatically enforced by the type system, but this is no longer true in (3). With this definition we finally write $\mathcal{E} \models_\gamma s \approx t$ to denote that the considered carriers are of type $\gamma\ set$.

By (3) it is straightforward to formalize Birkhoff's theorem using the proof ideas above. For the direction from semantic to syntactic entailment we choose γ to be the type of sets of terms, written $(\alpha, \beta)\ term\ set$ in Isabelle, where α is the type of function symbols and β the type of variables.

▶ **Theorem 6.1** (Birkhoff). *Let $\mathcal{E}$ be a set of equations over terms of type (α, β) term. Let α, β, and γ be arbitrary types. Then we have:*

- *Whenever $s \leftrightarrow_{\mathcal{E}}^* t$ then $\mathcal{E} \models_\gamma s \approx t$.*
- *Whenever $\mathcal{E} \models_{(\alpha,\beta)\ term\ set} s \approx t$ then $s \leftrightarrow_{\mathcal{E}}^* t$.*
- *$\mathcal{E} \models_{(\alpha,\beta)\ term\ set} s \approx t$ if and only if $s \leftrightarrow_{\mathcal{E}}^* t$.*

7 Conclusion

In this paper we formalized several properties of KBO including well-foundedness, where we used a variant with quasi-precedences, subterm coefficient functions, and infinite signatures. For well-foundedness we used a direct proof, and we illustrated why Kruskal's tree theorem could only be applied in a less powerful variant of KBO. Moreover, we have formalized the critical pair theorem, Birkhoff's theorem, and soundness of Knuth-Bendix completion for finite runs, where we could drop the strict encompassment condition. The latter result was already mentioned in a preliminary paper [22] and it has been extended in [28, Sections 5.1.2 and 6.1.2] to other versions of completion: Winkler showed that the strict encompassment condition can also be dropped for ordered completion [2] and normalized completion [15], if one considers finite runs, although without any formalization.

As far as we know, our results are the first formalizations of KBO, completion, and Birkhoff's theorem. Besides the generic theorems, we also developed executable criteria which allow us to certify completion proofs and (non-)derivability proofs, e.g., we can certify proofs from KBCV [23] and MKBTT [29].

Related work in Coq also allows checking a TRS for local confluence and termination, where C*i*ME3 [5] generates proof scripts for Coq. Although C*i*ME3 contains functionality to check derivability, we did not find any possibility to certify completion proofs or non-derivability proofs.

We also mention that resolution based ATPs are already used for finding and certifying proofs, but in a different way than we have presented: Sledgehammer [4] extracts the axioms used in a refutation proof (found by an external ATP) and then tries to find a certified proof using a variant of the Metis ATP whose inferences go through Isabelle's kernel.

An interesting future work would be to integrate our findings into Sledgehammer.

Acknowledgments

The authors are listed in alphabetical order regardless of individual contributions or seniority. We thank the anonymous referees for their helpful comments and remarks.

References

1 F. Baader and T. Nipkow. *Term Rewriting and All That*. New York, USA, paperback edition, Aug. 1999. doi:10.2277/0521779200.

2 L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. ACM*, 41(2):236–276, 1994. doi:10.1145/174652.174655.

3 G. Birkhoff. On the structure of abstract algebras. *Math. Proc. Cambridge*, 31(4):433–454, 1935. doi:10.1017/S0305004100013463.

4 S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In *International Joint Conference on Automated Reasoning, IJCAR'10*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. doi:10.1007/978-3-642-14203-1_9.

5 É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Automated certified proofs with C*i*ME3. In *Rewriting Techniques and Applications, RTA'11*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 21–30. doi:10.4230/LIPIcs.RTA.2011.21.

6 J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Inform.*, 28(2):95–119, 1990. doi:10.1007/BF01237233.

7 A. L. Galdino and M. Ayala-Rincón. A formalization of the Knuth-Bendix-Huet critical pair theorem. *J. Automat. Reason.*, 45(3):301–325, 2010. doi:10.1007/s10817-010-9165-2.

8 F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming, FLOPS'10*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. doi:10.1007/978-3-642-12251-4_9.

9 G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980. doi:10.1145/322217.322230.

10 D. E. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. 1970.

11 K. Korovin and A. Voronkov. Orienting rewrite rules with the Knuth-Bendix order. *Inform. and Comput.*, 183(2):165–186, 2003. doi:10.1016/S0890-5401(03)00021-X.

12 A. Krauss. Recursive definitions of monadic functions. In *Workshop on Partiality and Recursion in Interactive Theorem Proving, PAR'10*, volume 43 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–13. doi:10.4204/EPTCS.43.1.

13 A. Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Automat. Reason.*, 44(4):303–336, 2010. doi:10.1007/s10817-009-9157-2.

14 M. Ludwig and U. Waldmann. An extension of the Knuth-Bendix ordering with LPO-like properties. In *LPAR'07*, volume 4790 of *Lecture Notes in Computer Science*, pages 348–362. doi:10.1007/978-3-540-75560-9_26.

15 C. Marché. Normalized rewriting: An alternative to rewriting modulo a set of equations. *J. Symb. Comp.*, 21(3):253–288, 1996. doi:10.1006/jsco.1996.0011.

16 A. Middeldorp and H. Zantema. Simple termination of rewrite systems. *Theor. Comput. Sci.*, 175(1):127–158, 1997. doi:10.1016/S0304-3975(96)00172-7.

17 T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. 2002. doi:10.1007/3-540-45949-9.

18 J. Ruiz-Reina, J. Alonso, M. Hidalgo, and F.-J. Martín-Mateos. Formal proofs about rewriting using ACL2. *Ann. Math. Artif. Intel.*, 36(3):239–262, 2002. doi:10.1023/A:1016003314081.

19 J. Steinbach. Extensions and comparison of simplification orders. In *Rewriting Techniques and Applications, RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 434–448. doi:10.1007/3-540-51081-8_124.

20 C. Sternagel. A formal proof of Kruskal's tree theorem in Isabelle/HOL. draft.

21 C. Sternagel. Well-Quasi-Orders. *Archive of Formal Proofs*, Apr. 2012. http://afp.sourceforge.net/entries/Well_Quasi_Orders.shtml, Formal proof development.

22 T. Sternagel, R. Thiemann, H. Zankl, and C. Sternagel. Recording completion for finding and certifying proofs in equational logic. In *International Workshop on Confluence, IWC'12*, pages 31–36. Available at http://arxiv.org/abs/1208.1597.

23 T. Sternagel and H. Zankl. KBCV – Knuth-Bendix completion visualizer. In *International Joint Conference on Automated Reasoning, IJCAR'12*, volume 7364 of *Lecture Notes in Artificial Intelligence*, pages 530–536. doi:10.1007/978-3-642-31365-3_41.

24 R. Thiemann, G. Allais, and J. Nagele. On the formalization of termination techniques based on multiset orderings. In *Rewriting Techniques and Applications, RTA'12*, volume 15 of *Leibniz International Proceedings in Informatics*, pages 339–354. doi:10.4230/LIPIcs.RTA.2012.339.

25 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Theorem Proving in Higher Order Logics, TPHOLs'09*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. doi:10.1007/978-3-642-03359-9_31.

26 R. Thiemann and C. Sternagel. Loops under strategies. In *Rewriting Techniques and Applications, RTA'09*, volume 5595 of *Lecture Notes in Computer Science*, pages 17–31. doi:10.1007/978-3-642-02348-4_2.

27 Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *J. ACM*, 34(1):128–143, 1987. doi:10.1145/7531.7534.

28 S. Winkler. *Termination Tools in Automated Reasoning*. PhD thesis, University of Innsbruck, 2013.

29 S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara. Optimizing mkbTT (system description). In *Rewriting Techniques and Applications, RTA'10*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 373–384. doi:10.4230/LIPIcs.RTA.2010.373.

30 H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *J. Automat. Reason.*, 43(2):173–201, 2009. doi:10.1007/s10817-009-9131-z.

Automatic Decidability: A Schematic Calculus for Theories with Counting Operators

Elena Tushkanova[1,2], Christophe Ringeissen[1], Alain Giorgetti[1,2], and Olga Kouchnarenko[1,2]

1 **Inria, Villers-les-Nancy, F-54600, France**
Christophe.Ringeissen@loria.fr
2 **FEMTO-ST Institute (UMR 6174), University of Franche-Comté, Besançon, F-25030, France**
{elena.tushkanova,alain.giorgetti,olga.kouchnarenko}@femto-st.fr

Abstract

Many verification problems can be reduced to a satisfiability problem modulo theories. For building satisfiability procedures the rewriting-based approach uses a general calculus for equational reasoning named paramodulation. Schematic paramodulation, in turn, provides means to reason on the derivations computed by paramodulation. Until now, schematic paramodulation was only studied for standard paramodulation. We present a schematic paramodulation calculus modulo a fragment of arithmetics, namely the theory of Integer Offsets. This new schematic calculus is used to prove the decidability of the satisfiability problem for some theories equipped with counting operators. We illustrate our theoretical contribution on theories representing extensions of classical data structures, e.g., lists and records. An implementation within the rewriting-based Maude system constitutes a practical contribution. It enables automatic decidability proofs for theories of practical use.

1998 ACM Subject Classification F.4.1 Mechanical Theorem Proving, I.2.3 Inference Engines

Keywords and phrases decision procedures, superposition, schematic saturation

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.303

1 Introduction

Decision procedures for satisfiability modulo background theories of classical datatypes such as lists, arrays and records are at the core of many state-of-the-art verification tools. Designing and implementing these satisfiability procedures remains a very hard task. To help the researcher with this time-consuming task, an important approach based on rewriting has been investigated in the last decade [2, 1].

The rewriting-based approach allows building satisfiability procedures in a flexible way by using a general calculus for automated deduction, namely the paramodulation calculus [15] (also called superposition calculus). The paramodulation calculus is a refutation-complete inference system at the core of all equational theorem provers. In general this calculus provides a semi-decision procedure that halts on unsatisfiable inputs by generating an empty clause, but may not terminate on satisfiable ones. However, it also terminates on satisfiable inputs for some theories axiomatising standard datatypes such as arrays, lists, etc, and thus provides a decision procedure for these theories. A classical termination proof consists in considering the finitely many cases of inputs made of the (finitely many) axioms and any set of ground flat literals. This proof can be done by hand, by analysing the finitely many forms of clauses generated by saturation, but the process is tedious and error-prone. To

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 303–318

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


simplify this process, a schematic paramodulation calculus has been developed [11] to build the schematic form of the saturations. It can be seen as an abstraction of the paramodulation calculus: If it halts on one given abstract input, then the paramodulation calculus halts for all the corresponding concrete inputs. More generally, schematic paramodulation is a fundamental tool to check important properties related to decidability and combinability [12].

To ensure efficiency, it is very useful to have built-in axioms in the calculus, and so to design paramodulation calculi modulo theories. This is particularly important for arithmetic fragments due to the ubiquity of arithmetics in applications of formal methods. For instance, paramodulation calculi have been developed for Abelian Groups [10, 13] and Integer Offsets [14]. These calculi provide decision procedures for theories of practical interest.

New combination methods à la Nelson-Oppen have been designed for theories sharing these arithmetic fragments. This paves the way of using non-disjoint combination methods within SMT solvers. In [14], the termination of paramodulation modulo Integer Offsets is proved manually. Therefore, there is an obvious need for an automatic method to prove that an input theory admits a decision procedure based on paramodulation modulo Integer Offsets.

In this paper, we introduce theoretical underpinnings and a tool support that allow us to automatically prove the termination of paramodulation modulo Integer Offsets. We design a new schematic paramodulation calculus to describe saturations modulo Integer Offsets. Our approach requires a new form of schematization to cope with arithmetic expressions. The interest of schematic paramodulation relies on a correspondence between a derivation using (concrete) paramodulation and a derivation using schematic paramodulation: Roughly speaking, the set of derivations obtained by schematic paramodulation over-approximates the set of derivations obtained by (concrete) paramodulation. We show under which conditions the termination of schematic paramodulation implies the termination of (concrete) paramodulation. Again, the fact of considering Integer Offsets requires some specific proof arguments. We illustrate our contribution on the examples of theories considered in [14] – the theory of lists with length and the theory of records with increment. Our approach has been developed and validated thanks to a proof system we have implemented in the rewriting logic-based environment Maude [6] by using its reflection mechanism and its equational reasoning engines. This proof system implements schematic paramodulation modulo Integer Offsets. The proofs related to our examples are obtained via this automatic proof system.

Paper outline. Section 2 introduces preliminary notions related to first-order theories, the paramodulation and schematic paramodulation calculi, and theories with counting operators. Section 3 describes the new schematic paramodulation calculus and states conditions under which its termination implies the one of the (concrete) paramodulation calculus. Sections 4 and 5 respectively describe our implementation and experimentations with Integer Offsets extensions. Eventually, Section 6 concludes.

2 Background

We consider many-sorted first-order equational logic. A (many-sorted functional) *signature* Σ is a set of declarations of distinct function symbols and their type. A function declaration is of the form $f : s_1 \times \ldots \times s_n \to s$, where f is a function symbol, $n \geq 0$ is its arity, $s_1, \ldots, s_n$ and s are *sorts* from a finite set of sorts S. The sorts $s_1, \ldots, s_n$ are called the *argument sorts* and s is called the *value sort* of f. Each sort is interpreted by a nonempty domain. The only predicates are equalities on sorts, denoted $=_s$ for each sort $s \in S$, whose type is $s \times s$. We simply denote them $=$ when there is no risk of confusion.

Given a signature Σ, we assume the usual first-order syntactic notions of term, position, literal, clause, formula, substitution as defined, e.g., in [8]. When extended to many sorts, these notions additionally require that all terms have the appropriate sorts. A *rewrite system* is a set of directed equalities, called *rewrite rules*. It can be applied repeatedly along the direction given by the rules to replace equals by equals in any term. A term is in *normal form* if it cannot be rewritten any further. A rewrite system is *convergent* if its application to any term leads to a unique normal form.

Throughout this document, universally quantified variables are represented by capital letters. Given a term t and a position p, $t_{|p}$ denotes the subterm of t at position p, and $t[l]_p$ denotes the term t in which l appears as the subterm at position p. When the position p is clear from the context, we may simply write $t[l]$. Application of a substitution σ to a term t is written $\sigma(t)$. The notations $C[l]$ and $\sigma(C)$ are also used for any clause C. The *empty* clause, i.e. the clause with no disjunct, corresponding to an unsatisfiable formula, is denoted by $\bot$. The clause obtained from a clause C by replacing the terms occurring in C with their normal forms w.r.t. a convergent rewrite system R is denoted $C\downarrow_R$.

Terms, literals and clauses are *ground* whenever no variable appears in them. Given a function symbol f, an *f-rooted* term is a term whose top-symbol is f. A *compound* term is an f-rooted term for a function symbol f of positive arity. The *depth* of a term is defined inductively as follows: $depth(t) = 0$, if t is a constant or a variable, and $depth(f(t_1, \ldots, t_n)) = 1 + max\{depth(t_i) \mid 1 \leq i \leq n\}$. A term is *flat* if its depth is 0 or 1. A *positive literal* is an equality $l = r$ and a *negative literal* is a disequality $l \neq r$. We use the symbol $\bowtie$ to denote either $=$ or $\neq$. The depth of a literal $l \bowtie r$ is defined as follows: $depth(l \bowtie r) = depth(l) + depth(r)$. A positive literal is *flat* if its depth is 0 or 1. A negative literal is *flat* if its depth is 0.

We also assume the usual first-order notions of model, satisfiability, validity, logical consequence. A *first-order theory* (over a finite signature) is a set of first-order formulae with no free variables. When T is a finitely axiomatized theory, $Ax(T)$ denotes the set of axioms of T. In this paper we consider first-order theories *with equality*, for which the equality symbol $=$ is always interpreted as the equality relation. A formula is *satisfiable in a theory* T if it is satisfiable in a model of T. The *satisfiability problem* modulo a theory T amounts to establishing whether any given finite conjunction of literals (or equivalently, any given finite set of literals) is T-satisfiable or not.

We consider inference systems using well-founded orderings on terms (resp. literals) that are total on ground terms (resp. literals). An ordering $<$ on terms is a *simplification ordering* [8] if it is stable ($l < r$ implies $\sigma(l) < \sigma(r)$ for every substitution σ), monotonic ($l < r$ implies $t[l]_p < t[r]_p$ for every term t and position p), and has the subterm property (i.e., it contains the subterm ordering: if l is a strict subterm of r, then $l < r$). Simplification orderings are well-founded for finite signatures. A term t is *maximal* in a multiset S of terms if there is no $u \in S$ such that $t < u$. Hence, if $t \not\leq u$, then t and u are different terms and t is maximal in $\{t, u\}$. An ordering on terms is extended to literals by using its multiset extension on literals viewed as multisets of terms. Any positive literal $l = r$ (resp. negative literal $l \neq r$) is viewed as the multiset $\{l, r\}$ (resp. $\{l, l, r, r\}$). Also, a term is *maximal* in a literal whenever it is maximal in the corresponding multiset.

2.1 Paramodulation Calculus

As in [16] we consider only unitary clauses, i.e. clauses composed of at most one literal. The *Unitary Paramodulation Calculus* is the inference system $\mathcal{UPC}$ [16] consisting of the rules in Figures 1 and 2, where $\cup$ stands for disjoint union. Expansion rules (Figure 1) aim at generating new (deduced) literals. The *Superposition* rule uses an equality to perform

a replacement of equal by equal into a literal. The *Reflection* rule generates the empty clause when both sides of a disequality are unifiable. Contraction rules (Figure 2) aim at simplifying the set of literals. Using *Subsumption*, a literal is removed when it is an instance of another one. *Simplification* rewrites a literal into a simpler one by using an equality that can be considered as a rewrite rule. *Deletion* removes trivial equalities. We assume the usual definitions of redundancy, saturation, derivation and fairness as defined, e.g., in [16].

A fundamental feature of $\mathcal{UPC}$ is the usage of a simplification ordering $<$ to control the application of *Superposition* and *Simplification* rules by orienting equalities. Hence, the *Superposition* rule is applied by using terms that are maximal in their literals with respect to $<$. This ordering is total on ground terms. We use a lexicographic path ordering [8].

Let us recall the usual definitions of redundancy, saturation, derivation and fairness. A clause C is *redundant* with respect to a set S of clauses if either $C \in S$ or S can be obtained from $S \cup \{C\}$ by a sequence of applications of contraction rules (cf. Figure 2). An inference is *redundant* with respect to a set S of clauses if its conclusion is redundant with respect to S. A set S of clauses is *saturated* if every inference with a premise in S is redundant with respect to S. A *derivation* is a sequence $S_0, S_1, \ldots, S_i, \ldots$ of sets of clauses where each S_{i+1} is obtained from S_i by applying an inference to add a clause (by expansion rules in Figure 1) or to delete a clause (by contraction rules in Figure 2). One can remark that an application of the *Simplification* rule corresponds to two steps in the derivation: the first step adds a new literal, whilst the second one deletes a literal. A derivation is characterized by its *limit*, defined as the set of persistent clauses $\bigcup_{j\geq 0}\bigcap_{i>j} S_i$, that is, the union for each $j \geq 0$ of the set of clauses occurring in all future steps starting from S_j. A derivation $S_0, S_1, ..., S_i, ...$ is *fair* if for every inference with premises in the limit, there is some $j \geq 0$ such that the inference is redundant with respect to S_j. The set of persistent literals obtained by a fair derivation is called the *saturation* of the derivation.

Superposition $\dfrac{l[u'] \bowtie r \qquad u = t}{\sigma(l[t] \bowtie r)}$

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not a variable.

Reflection $\dfrac{u' \neq u}{\bot}$

Above, u and u' are unifiable and σ is the most general unifier of u and u'.

Figure 1 Expansion inference rules of $\mathcal{UPC}$.

Subsumption $\dfrac{S \cup \{L, L'\}}{S \cup \{L\}}$ **if** $L' = \sigma(L)$.

Simplification $\dfrac{S \cup \{C[l'], l = r\}}{S \cup \{C[\sigma(r)], l = r\}}$

if i) $l' = \sigma(l)$, ii) $\sigma(l) > \sigma(r)$, and iii) $C[l'] > (\sigma(l) = \sigma(r))$.

Deletion $\dfrac{S \cup \{u = u\}}{S}$

Figure 2 Contraction inference rules of $\mathcal{UPC}$.

2.2 Paramodulation Calculus for Integer Offsets

The paramodulation calculus defined in [14] adapts the calculus $\mathcal{UPC}$ to the theory of Integer Offsets, so that it can serve as a basis for the design of decision procedures for Integer Offsets extensions. The theory of Integer Offsets is axiomatized by the following set of axioms:

$$\begin{array}{lll} (s0) & \forall X. & \mathsf{s}(X) \neq \mathsf{0} \\ (inj) & \forall X, Y. & \mathsf{s}(X) = \mathsf{s}(Y) \Rightarrow X = Y \\ (acy) & \forall X. & X \neq \mathsf{s}^n(X) \quad \text{for all} \quad n \geq 1 \end{array}$$

over the signature $\Sigma_I := \{\mathsf{0} : \textsc{int}, \mathsf{s} : \textsc{int} \to \textsc{int}\}$. The second axiom specifies that the successor function s is injective. The third axiom is in fact an axiom scheme, which specifies that this function is acyclic.

A (non-disjoint) *Integer Offsets extension* is a many-sorted theory whose set of sorts contains INT, whose signature shares symbols with Σ_I, and whose axioms possibly involve the symbols s and $\mathsf{0}$. Following [14, Section 5], we consider two Integer Offsets extensions: the theory of lists with length whose signature is $\Sigma_{LLI} = \{\mathsf{car} : \textsc{lists} \to \textsc{elem}, \mathsf{cdr} : \textsc{lists} \to \textsc{lists}, \mathsf{cons} : \textsc{elem} \times \textsc{lists} \to \textsc{lists}, \mathsf{len} : \textsc{lists} \to \textsc{int}, \mathsf{nil} :\to \textsc{lists}, \mathsf{0} :\to \textsc{int}, \mathsf{s} : \textsc{int} \to \textsc{int}\}$ and whose set of axioms $Ax(LLI)$ consists of

$$\begin{array}{ll} \mathsf{car}(\mathsf{cons}(X, Y)) = X & \mathsf{cons}(X, Y) \neq \mathsf{nil} \\ \mathsf{cdr}(\mathsf{cons}(X, Y)) = Y & \mathsf{len}(\mathsf{nil}) = \mathsf{0} \\ \mathsf{len}(\mathsf{cons}(X, Y)) = \mathsf{s}(\mathsf{len}(Y)) & \end{array}$$

where X is a universally quantified variable of sort ELEM, and Y is a universally quantified variable of sort LISTS, and the theory of records of length 3 (without extensionality) with increment whose signature is $\Sigma_{RII} = \bigcup_{i=1}^{3} \{\mathsf{rstore}_i : \textsc{rec} \times \textsc{int} \to \textsc{rec}, \mathsf{rselect}_i : \textsc{rec} \to \textsc{int}, \mathsf{incr} : \textsc{rec} \to \textsc{rec}, \mathsf{s} : \textsc{int} \to \textsc{int}\}$ and whose set of axioms $Ax(RII)$ consists of

$$\begin{array}{rcl} \mathsf{rselect}_i(\mathsf{rstore}_i(X, Y)) & = & Y \text{ for } i \in \{1, 2, 3\} \\ \mathsf{rselect}_i(\mathsf{rstore}_j(X, Y)) & = & \mathsf{rselect}_i(X) \text{ for } i, j \in \{1, 2, 3\} \text{ with } i \neq j \\ \mathsf{rselect}_i(\mathsf{incr}(X)) & = & \mathsf{s}(\mathsf{rselect}_i(X)) \text{ for } i \in \{1, 2, 3\} \end{array}$$

where X is a universally quantified variable of sort REC, and Y is a universally quantified variable of sort INT.

R1 $\quad \dfrac{S \cup \{\mathsf{s}(u) = \mathsf{s}(v)\}}{S \cup \{u = v\}}$ **if** u and v are ground terms.

R2 $\quad \dfrac{S \cup \{\mathsf{s}(u) = t, \mathsf{s}(v) = t\}}{S \cup \{\mathsf{s}(v) = t, u = v\}}$

if u, v and t are ground terms, $\mathsf{s}(u) > t$, $\mathsf{s}(v) > t$ and $u > v$.

C1 $\quad \dfrac{S \cup \{\mathsf{s}(t) = 0\}}{S \cup \{\mathsf{s}(t) = 0, \bot\}}$ **if** t is a ground term.

C2 $\quad \dfrac{S \cup \{\mathsf{s}^n(t) = t\}}{S \cup \{\mathsf{s}^n(t) = t, \bot\}}$ **if** t is a ground term and $n \geq 1$.

Figure 3 Ground reduction rules for Integer Offsets.

The paramodulation calculus $\mathcal{UPC}_I$ consists of the expansion rules of $\mathcal{UPC}$ (Figure 1), the contraction rules of $\mathcal{UPC}$ (Figure 2) plus some additional reduction rules corresponding

to the axioms of the theory of Integer Offsets (Figure 3). We use a $T_I - \mathit{good}$ ordering $<$ to control $\mathcal{UPC}_I$: $<$ is a simplification ordering which is total on ground terms, such that $\mathsf{0}$ is minimal and, for any non s-rooted terms t_1 and t_2, $\mathsf{s}^{n_1}(t_1) > \mathsf{s}^{n_2}(t_2)$ iff either $t_1 = t_2$ and n_1 is bigger than n_2, or $t_1 > t_2$. The refutation completeness of $\mathcal{UPC}_I$ is studied in [14].

2.3 Schematic Paramodulation

The motivation of schematic paramodulation is to derive a schematic form of saturations computed by paramodulation. In the following, we consider the *Schematic Unitary Paramodulation Calculus*, denoted by $\mathcal{SUPC}$, as an abstraction of $\mathcal{UPC}$. Indeed, any concrete saturation computed by $\mathcal{UPC}$ can be viewed as an instance of an abstract saturation computed by $\mathcal{SUPC}$ [12, Theorem 2]. Hence, if $\mathcal{SUPC}$ halts on one given abstract input, then $\mathcal{UPC}$ halts for all the corresponding concrete inputs. More generally, $\mathcal{SUPC}$ is an automated tool to check properties of $\mathcal{UPC}$ such as termination, stable infiniteness and deduction completeness [12].

$\mathcal{SUPC}$ is almost identical to $\mathcal{UPC}$, except that literals are constrained by conjunctions of atomic constraints of the form $\mathit{const}(x)$ where x is a variable. An implementation of *Paramodulation* and *Schematic Paramodulation* calculi $\mathcal{UPC}$ and $\mathcal{SUPC}$ is presented in [16].

Let us recall the notions of constrained clause and instance of constrained clause used in [12] for $\mathcal{SUPC}$. An *atomic constraint* is of the form $\mathit{const}(t)$, where t is a term. It is true iff t is a constant. A *constraint* is a conjunction of atomic constraints which is true if each atomic constraint in the conjunction is true. For sake of brevity, $\mathit{const}(t_1, \ldots, t_n)$ denotes the conjunction $\mathit{const}(t_1) \wedge \cdots \wedge \mathit{const}(t_n)$. A *constrained clause* is of the form $C \parallel \varphi$, where C is a clause and φ is a constraint. A variable x is *constrained* in a constrained clause $C \parallel \varphi$ if $\mathit{const}(x)$ is in φ; otherwise it is *unconstrained*. We say that $\sigma(C)$ is a *constraint instance* of $C \parallel \varphi$ if the domain of σ contains all the constrained variables in $C \parallel \varphi$, the range of σ contains only constants and $\sigma(\varphi)$ is satisfiable. By a slight abuse of notation, we say that a term u is *ground* with respect to a constraint φ if all the variables in u are constrained (are in φ).

3 Schematic Paramodulation Calculus for Integer Offsets

This section introduces a new schematic calculus denoted by $\mathcal{SUPC}_I$. It is a schematization of $\mathcal{UPC}_I$ taking into account the axioms of the theory of Integer Offsets within the framework based on schematic paramodulation introduced in Section 2.3.

The theory of Integer Offsets allows us to build arithmetic expressions of the form $\mathsf{s}^n(t)$ for $n > 0$. The idea investigated here is to represent all terms of this form in a unique way. To this end, we consider a new operator $\mathsf{s}^+ : \text{INT} \to \text{INT}$ such that $\mathsf{s}^+(t)$ denotes the infinite set of terms $\{\mathsf{s}^n(t) \mid n \geq 1\}$. A *schematic term* is a term containing s^+.

The calculus $\mathcal{SUPC}_I$ handles schematic clauses that extend constrained clauses of $\mathcal{SUPC}$.

▶ Definition 1 (Schematic Clause). A *schematic clause* is a constrained clause built over the signature extended with s^+. An *instance of a schematic clause* is a constraint instance where each occurrence of s^+ is replaced by some s^n with $n > 0$.

The calculus $\mathcal{SUPC}_I$ takes as input a set of schematic literals, G_0, that represents all possible sets of ground literals given as inputs to $\mathcal{UPC}_I$:

$$\begin{aligned} G_0 = \quad & \{\bot, x = y \parallel \mathit{const}(x, y), x \neq y \parallel \mathit{const}(x, y), u = \mathsf{s}^+(v) \| \varphi\} \\ & \cup \textstyle\bigcup_{f \in \Sigma_T} \{f(x_1, \ldots, x_n) = x_0 \parallel \mathit{const}(x_0, x_1, \ldots, x_n)\} \end{aligned}$$

where u, v are flat terms of sort INT whose variables are all constrained, and x, y are constrained variables of the same sort.

3.1 Schematic Calculus

The calculus $\mathcal{SUPC}_I$ is depicted in Figures 4, 5 and 6. It re-uses most of the rules of $\mathcal{SUPC}$ (Figures 4 and 5) and completes them with two rules (Figure 6) coming from the reduction rules for Integer Offsets (Figure 3).

In [14] the definition of *derivation* has been adapted to the paramodulation calculus for Integer Offsets. Similarly, we adapt the standard definition of derivation to the schematic paramodulation calculus modulo Integer Offsets.

▶ Definition 2. A derivation with respect to $\mathcal{SUPC}_I$ is a (finite or infinite) sequence of sets of literals $S_1, S_2, S_3, \ldots, S_i, \ldots$ such that, for every i, it holds that:

1. S_{i+1} is obtained from S_i by adding a literal obtained by the application of one of the rules in Figures 4, 5 and 6 to some literals in S_i;
2. S_{i+1} is obtained from S_i by removing a literal according to one of the rules in Figures 5 and 6.

Superposition $\dfrac{l[u'] \bowtie r \| \varphi \qquad u = t \| \psi}{\sigma(l[t] \bowtie r \| \varphi \wedge \psi)}$

if i) $\sigma(u) \not\leq \sigma(t)$, ii) $\sigma(l[u']) \not\leq \sigma(r)$, and iii) u' is not an unconstrained variable.

Reflection $\dfrac{u' \neq u \| \psi}{\bot}$ **if** $\sigma(\psi)$ is satisfiable.

Above, u and u' are unifiable and σ is the most general unifier of u and u'.

Figure 4 Schematic expansion rules.

We use a specific term rewrite system to simplify schematic terms. Hence, the rewrite system $R\mathsf{s}^+ = \{\ \mathsf{s}^+(\mathsf{s}(x)) \to \mathsf{s}^+(x),\ \mathsf{s}(\mathsf{s}^+(x)) \to \mathsf{s}^+(x),\ \mathsf{s}^+(\mathsf{s}^+(x)) \to \mathsf{s}^+(x)\ \}$ is applied eagerly whenever a new literal is generated by superposition or simplification. For each of these rules, one can easily check that the set of terms denoted by the left-hand side is included in the set of terms denoted by the right-hand side.

Let us notice that the two rules $C1$ and $C2$ from $\mathcal{UPC}_I$ (Figure 3) do not appear in $\mathcal{SUPC}_I$. This is due to the fact that these rules produce only the empty clause $\bot$ which occurs already in the input set G_0. Note that the *Reflection* rule could be omitted as well, but it is kept because it cannot be removed in the non-unitary case.

Similarly to [12], we introduce a specific *Schematic Deletion* rule to avoid the divergence of $\mathcal{SUPC}_I$. Let us motivate this rule by considering the theory of lists with length. In fact, the calculus generates a schematic clause $\mathsf{len}(a) = \mathsf{s}(\mathsf{len}(b)) \| const(a, b)$ which will superpose with a renamed copy of itself, i.e. with $\mathsf{len}(a') = \mathsf{s}(\mathsf{len}(b'))\ \| const(a', b')$ to generate a schematic clause of a new form $\mathsf{len}(a) = \mathsf{s}(\mathsf{s}(\mathsf{len}(b')))\ \| const(a, b')$. This process continues to generate deeper and deeper schematic clauses so that the Schematic Saturation will diverge. To avoid this divergence problem, the additional *Schematic Deletion* rule aims at checking whether instances of a schematic clause C' are instances of another schematic clause C containing an occurrence of s^+. To implement this rule, we apply a morphism π replacing all the occurrences of s in C' by s^+ ($\pi(\mathsf{s}(t)) = \mathsf{s}^+(\pi(t))$ for any t, $\pi(x) = x$ if x is a variable), and then the rewrite system $R\mathsf{s}^+$ to replace a chain of s^+ occurrences in $\pi(C')$ by only one occurrence of

Subsumption $\dfrac{S \cup \{L\|\psi, L'\|\psi'\}}{S \cup \{L\|\psi\}}$

if either a) $L \in Ax(T)$, ψ is empty and for some substitution σ, $L' = \sigma(L)$; or b) $L' = \sigma(L)$ and $\psi' = \sigma(\psi)$, where σ is a renaming or a mapping from constrained variables to constrained variables.

Simplification $\dfrac{S \cup \{C[l']\|\varphi, l = r\}}{S \cup \{C[\sigma(r)]\|\varphi, l = r\}}$

if i) $l = r \in Ax(T)$, ii) $l' = \sigma(l)$, iii) $\sigma(l) > \sigma(r)$, and iv) $C[l'] > (\sigma(l) = \sigma(r))$.

Tautology $\dfrac{S \cup \{u = u\|\varphi\}}{S}$

Deletion $\dfrac{S \cup \{L\|\varphi\}}{S}$ **if** φ is unsatisfiable.

Schematic Del. $\dfrac{S \cup \{C'\|\varphi, C[\mathsf{s}^+(t)]\|\psi\}}{S \cup \{C[\mathsf{s}^+(t)]\|\psi\}}$

if $\sigma(\pi(C') \downarrow_{R\mathsf{s}^+}) = C[\mathsf{s}^+(t)]$, $\sigma(\varphi) = \psi$, for a renaming σ.

Figure 5 Schematic contraction rules.

R1 $\dfrac{S \cup \{\mathsf{s}(u) = \mathsf{s}(v)\|\varphi\}}{S \cup \{u = v\|\varphi\}}$ **if** u and v are ground terms.

R2 $\dfrac{S \cup \{\mathsf{s}(u) = t\|\varphi, \mathsf{s}(v) = t\|\psi\}}{S \cup \{\mathsf{s}(v) = t\|\psi, u = v\|\psi \wedge \varphi\}}$

if u, v and t are ground terms, $\mathsf{s}(u) > t$, $\mathsf{s}(v) > t$ and $u > v$.

Figure 6 Ground reduction rules.

s^+. For instance, *Schematic Deletion* rule applies for the theory of lists with length since G_0 already contains the non-flat schematic literal $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b))\|const(a, b)$, and so it subsumes a schematic clause like $\mathsf{len}(a) = \mathsf{s}(\mathsf{s}(\mathsf{len}(b'))) \ \|const(a, b')$. Similarly, the schematic saturation of the theory of records with increment diverges. But thanks to the *Schematic Deletion* rule, it terminates since the initial set of schematic literals additionally contains the schematic literal $\mathsf{rselect}_i(a) = \mathsf{s}^+(\mathsf{rselect}_i(b))\|const(a, b)$. A generalization of this condition to any theory extending Integer Offsets consists in adding to the standard definition of G_0 (introduced in [12]) a finite set of the schematic literals of the form $u = \mathsf{s}^+(v)\|\varphi$, where u and v are two flat terms of sort INT whose variables are constrained by φ.

3.2 Adequation Result

We show that any clause in a saturation obtained by $\mathcal{UPC}_I$ is an instance of a schematic clause in a schematic saturation obtained by $\mathcal{SUPC}_I$, under the following assumption.

▶ Assumption 1. Let SC be any set of schematic clauses generated by $\mathcal{SUPC}_I$. If an s^+-rooted term (resp. s-rooted term) occurs in a term u which is maximal in an equality $u = t$ in SC, then there is no s-rooted term (resp. s^+-rooted term) occurring in a term $l[u']$ which is maximal in a clause $l[u'] \bowtie r$ in SC.

Without Assumption 1 we would need a specific unification algorithm to handle literals involving s and s^+. Thanks to it we can continue applying the superposition rule with syntactic unification.

▶ **Theorem 1.** *Let T be a theory axiomatized by a finite set $Ax(T)$ of literals, which is saturated with respect to $\mathcal{UPC}_I$. Let G_∞ be the set of all schematic clauses in a saturation of $Ax(T) \cup G_0$ by $\mathcal{SUPC}_I$. Then for every set S of ground flat literals, every clause in a saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$ is an instance of a schematic clause in G_∞.*

Proof. The proof is an adaptation of the one of [12, Theorem 2]. The proof is by induction on the length of derivations of $\mathcal{UPC}_I$. The base case is obvious. For the inductive case, we need to show two facts:

(1) each clause added in the process of saturation of $Ax(T) \cup S$ is an instance of a schematic clause in the saturation G_∞ of $Ax(T) \cup G_0$ by $\mathcal{SUPC}_I$, and

(2) if a clause is deleted by *Subsumption*, *Tautology* or *Deletion* from (or simplified by *Simplification*/reduced by *Reduction* in) G_∞, then all instances of the latter will also be deleted from (or simplified/reduced in) the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$.

Moreover, because of additional rewriting rules for terms containing s^+, we have to check another fact:

(3) Any such rule preserves the set of instances of any schematic clause.

Proof of (1). Consider the *Superposition* rule of $\mathcal{UPC}_I$. By induction hypothesis $l[u'] \bowtie r$ and $u = t$ are instances of schematic clauses in G_∞, i.e. there is some schematic clause $\hat{D}$ in G_∞ such that $l[u'] \bowtie r$ is an instance of $\hat{D}$, and a schematic clause $\hat{E}$ in G_∞ such that $u = t$ is an instance of $\hat{E}$. Two cases can be distinguished:

(∗) If there is no occurrence of s in u or u', then there exists a *Superposition* inference of $\mathcal{SUPC}_I$ in G_∞ whose premises are $\hat{D}$ and $\hat{E}$, and whose conclusion is a schematic clause $\hat{C}$ such that $\sigma(l[t] \bowtie r)$ is an instance of $\hat{C}$, where σ denotes the most general unifier of u and u'.

(∗∗) If there are occurrences of s in both u and u', two additional subcases can be considered. Assume that $\hat{u}$ and $\hat{u'}$ denote the schematic terms of u and u'.

1. If $\hat{u}$ and $\hat{u'}$ contain only s^+-rooted terms (resp. s-rooted terms), then we proceed as in (∗).
2. If $\hat{u}$ contains an s^+-rooted term (resp. s-rooted term) and $\hat{u'}$ contains an s-rooted term (resp. s^+-rooted term), then $\hat{u}$ may not unify with $\hat{u'}$ since we use syntactic unification, while u and u' may unify. This subcase is avoided by Assumption 1 and the side conditions of the *Superposition* rule.

Reflection of $\mathcal{UPC}_I$ can be handled in a way similar to *Superposition* and is therefore omitted.

Proof of (2). Let us consider *Subsumption* of $\mathcal{SUPC}_I$. For the case (a), let us assume that there are a schematic clause A deleted from G_∞ and a clause B in the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$, which is an instance of the schematic clause A. Then there must exist a clause $C \in Ax(T)$ and some substitution θ such that $\theta(C) \subseteq A$. Since all the clauses in $Ax(T)$ persist, there must be a substitution θ' such that $\theta'(C) \subseteq B$. Thereby B must also be deleted from the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$, and we are done. The case (b) of *Subsumption* is just a matter of deleting duplicates and leaving only more general constrained literals.

Since axioms do not contain the s^+ symbol, a similar argument can be used for *Simplification* of $\mathcal{SUPC}_I$. Assume that there is a schematic clause $C[l'] \parallel \varphi$ in G_∞ simplified by an equality $l = r$ ($l = r \in Ax(T)$) into $C[\theta(r)] \parallel \varphi$. Let σ be a substitution such that $\sigma(C[l'])$ is an instance of $C[l'] \parallel \varphi$. Since $l = r$ persists in the saturation of $Ax(T) \cup S$ by $\mathcal{UPC}_I$, there must be a simplification of $\sigma(C[l']) = \sigma(C)[\sigma(\theta(l))]$ by $l = r$ into $\sigma(C)[\sigma(\theta(r))] = \sigma(C[\theta(r)])$, which is an instance of $C[\theta(r)] \parallel \varphi$.

For the *Tautology Deletion* rule of $\mathcal{SUPC}_I$, it is easy to see that a constraint instance of a tautology is also a tautology. For the *Deletion* rule of $\mathcal{SUPC}_I$, notice that clauses with an unsatisfiable constraint have no instances.

For the reduction rule $R1$ of $\mathcal{SUPC}_I$, it is easy to see that an instance of a schematic clause $\mathsf{s}(u) = \mathsf{s}(v)$ will also reduce a root symbol s. For the reduction rule $R2$ of $\mathcal{SUPC}_I$, a similar argument can be given.

Proof of (3). Let $C \downarrow_{R\mathsf{s}^+}$ be the clause obtained from C by replacing the terms occurring in C with their normal forms w.r.t. $R\mathsf{s}^+$. The set of (concrete) clauses schematized by a schematic clause C is included in the set of (concrete) clauses schematized by $C \downarrow_{R\mathsf{s}^+}$, because a similar inclusion holds for all the terms in C and all the rules in $R\mathsf{s}^+$. ◀

3.3 Application to the Termination of Paramodulation

Contrary to the standard case, a schematized saturation may represent an infinite set of clauses since the term $\mathsf{s}^+(t)$ represents all the terms $s^n(t)$ with $n \geq 1$. The difficulty is then to prove the termination in this case. In [14], the termination proofs do not only rely on the fact that there are finitely many forms of clauses generated by the paramodulation calculus. In addition, the following proof argument is used: any new ground literal is strictly smaller than the biggest ground literal in the input set. Similarly, whereas the schematic paramodulation allows computing the different forms of clauses generated by paramodulation, we still need an additional analysis to conclude that the paramodulation calculus terminates. Fortunately, this analysis can be easily performed for some cases. We investigate hereafter a new solution where the analysis is restricted to the (few) schematic equalities containing s^+ that occur in the (finite) schematic saturation.

▶ Assumption 2. A schematic equality containing s^+ cannot be instantiated with different values of the exponent of s in a saturation of a satisfiable input.

Thanks to Assumption 2, there are only finitely many possible instances in the saturation of a satisfiable input. For instance, we cannot have both $i = \mathsf{s}(j)$ and $i = \mathsf{s}^2(j)$ in the saturation of a satisfiable input due to the acyclicity axiom.

With respect to disequalities, we restrict us to the case where $\mathcal{UPC}_I$ does not generate new disequalities having an occurrence of s: the simplification of an input disequality is the only way to have a disequality with an occurrence of s introduced in a derivation with $\mathcal{UPC}_I$.

This restriction is satisfied in the following cases:

- The set of axioms of the theory contains only equalities.
- The set of axioms of the theory contains some disequalities of a given sort, say D, such that it is not possible to build terms of sort D containing s.

Hence, this restriction is satisfied for the theories we are interested in.

▶ **Theorem 2.** *Assume that $\mathcal{UPC}_I$ does not generate new disequalities having an occurrence of* s*. If $\mathcal{SUPC}_I$ generates a finite schematic saturation such that all its schematic equalities satisfy Assumption 2, then $\mathcal{UPC}_I$ terminates on any input set of ground literals.*

Proof. Consider a satisfiable input. Let n_d (resp. n_e) be the number of disequalities (resp. equalities) obtained from the schematic disequalities (resp. equalities) in the finite schematic saturation by considering all possible instantiations of constrained variables by the finitely many constants in the input. The number of clauses occurring in the saturation of the input can be bounded as follows:

1. By hypothesis, the number of disequalities cannot be greater than $n_d + i_d$, where i_d denotes the number of disequalities in the input set.
2. Consider the equalities. According to Assumption 2, the number of equalities is bounded by n_e.

Consequently, the paramodulation calculus computes a finite saturated set of clauses and terminates. ◀

One can remark that our restriction on the generation of new disequalities in Theorem 2 is expressed with $\mathcal{UPC}_I$, but not with $\mathcal{SUPC}_I$. We adopt this solution because this restriction is easy to satisfy in practice and it is sufficient for our need. Of course, an interesting problem would be to find a way at the schematic level to ensure the boundedness of the generated disequalities. A possible solution could be envisioned when all the generated schematic literals are ground.

4 Implementation

This section presents an implementation of the schematic paramodulation calculus modulo Integer Offsets $\mathcal{SUPC}_I$, by using the Maude system [7] and its support of rewriting logic. It extends an implementation of $\mathcal{SUPC}$ described in [16]. We reuse the expansion and contraction rules implemented in [16]. The new implementation supports many-sorted theories. We discuss the normalization of schematic terms containing s^+, the implementation of the new reduction rules and the *Schematic Deletion* rule. We briefly introduce our support for derivation traces before showing our experimental results.

Sorts. The underlying logic of Maude is order-sorted, admitting a subsort ordering, whereas the underlying logic of our calculus $\mathcal{SUPC}_I$ is many-sorted, i.e. there is no subsort relation between sorts in the addressed theories. Let Σ be a many-sorted signature and S be its set of sorts. When implementing Σ in Maude, each sort in S is implemented as a Maude sort. For the theory of lists with length, the sorts LISTS, ELEM and INT are implemented by the declaration

```
sorts Lists Elem Ints .
```

in Maude.[1] Moreover, no subsort relation is declared between these Maude sorts. This condition guarantees that the order-sorted features of Maude (pattern-matching, unification, etc) behave as many-sorted ones on the set of Maude sorts associated to S.

Schematic Literals. We take profit of the powerful reflection mechanism of Maude. Maude terms are reflected as "meta-terms" with sort `Term`. The base cases in the metarepresentation of terms are given by the subsorts `Constant` and `Variable` of the sort `Term`.

Most of our implementation works at the meta-level, i.e. its functions operate on meta-terms with sort `Term`. Literals are defined by

```
sort Literal .
op _equals_ : Term Term -> Literal [comm] .
op _!=_     : Term Term -> Literal [comm] .
```

The attribute `[comm]` declares that the infix binary symbols `equals` and `!=` for equality and disequality are commutative. Then, the sort `SLiteral` of schematic literal is declared by

```
sorts SLiteral .
op emptyClause : -> SLiteral .
op ax : Literal -> SLiteral .
op _ || _ : Literal Constraint -> SLiteral .
```

where the infix operator `||` constructs a constrained literal from a literal and a constraint of sort `Constraint`. A constraint is implemented as a set of atomic constraints of the form `const(`t`)` where t is a term. An atomic constraint is satisfiable iff t is of subsort `Variable`, but unification sometimes produces `const(`t`)` where t is not a variable. Such a constraint is afterwards detected as unsatisfiable.

Normalization of Schematic Terms. The rewrite system $R\mathsf{s}^+$ is convergent, i.e. it computes a unique normal form. The `nf` function (`nf : Term -> Term`) computes this normal form. This function is applied eagerly whenever a new literal is generated. The normalization of terms is extended to literals by the `nfLit` function (`nfLit : Literal -> Literal`) that normalizes both sides of a given literal.

Ground Reduction Rules. Let us now present the encoding of the reduction rules of $\mathcal{SUPC}_I$. We translate them into rewrite rules.

The *R1* reduction rule is encoded by the following conditional rewrite rule:

```
crl [red1] :
 'succ[U] equals 'succ[U'] || Phi => U equals U' || Phi
   if isVarInConstraint(U, Phi) and isVarInConstraint(U', Phi) .
```

This rule removes the root symbol in both sides of a literal if this root symbol is `'succ` (`'succ` stands for `s`) and the variables of their subterms `U` and `U'` are constrained (are in `Phi`). This condition is checked by the function `isVarInConstraint`.

The following Maude conditional rewrite rule encodes the *R2* reduction rule.

```
crl [red2] :
'succ[U] equals T || Phi1, 'succ[V] equals T || Phi2 =>
'succ[V] equals T || Phi2, U equals V || cleanConstraint(U, V, Phi1, Phi2)
```

[1] An `s` is added to the Maude sort names for integers and lists because Maude sorts named `Int` and `List` already exist.

```
if isVarInConstraint(U, Phi1) and isVarInConstraint(V, Phi2) and
   isVarInConstraint(T, Phi1) and gtLPO('succ[U], Phi1, T) and
   gtLPO('succ[V], Phi2, T) and gtLPO(U, (Phi1, Phi2), V) .
```

The ordering $>$ on terms is implemented as a Boolean function `gtLPO` such that `gtLPO`$(u,$ `Phi`$, t)$ $=$ `true` iff $u > t$. We add the additional parameter `Phi` that collects the constrained variables, since constrained variables of different sorts are comparable. The function `cleanConstraint` aims at removing the constrained variables that do no occur in $u = v$.

Schematic Deletion. The *Schematic Deletion* rule is encoded by the following Maude conditional rewrite rule

```
crl [sdel] : L || Phi1, L' || Phi2 => L' || Phi2
  if conditionDel(L || Phi1, L' || Phi2) .
```

The function `conditionDel` implements the side conditions of the *Schematic Deletion* rule presented in Figure 5.

Traces. An additional and important feature of our tool consists in providing a trace indicating the name of the applied rule, the schematic clauses it is applied to at each derivation step and the position at which the rule is applied. This trace helps understanding the origin of each new schematic clause.

Let us show the syntax of traces for the superposition rule:

$$\frac{l[u'] \bowtie r \| \varphi \qquad u = t \| \psi}{\sigma(l[t] \bowtie r \| \varphi \wedge \psi)}$$

The expression `sup(`C_1`,` C_2`,` u`,` $l[u']$`,` Ctx`)` `gives` C_3 means that the constrained clause $C_3 = \sigma(l[t] \bowtie r \| \varphi \wedge \psi)$ is derived from the constrained clauses $C_1 = (l[u'] \bowtie r \| \varphi)$ and $C_2 = (u = t \| \psi)$ by superposing term u from C_2 in term $l[u']$ from C_1 at the context $Ctx = l[\,]$, where the rewriting has taken place.

5 Experimental Results

The implementation has been used to compute the schematic saturations for the theories LLI and RII introduced in Section 2.2. These computations generate the expected results, as shown in [17].

5.1 Example 1

Consider $Ax(LLI) \cup G_0$ that consists of the empty clause and the following literals:

1. Axioms for lists
   a) $\mathsf{car}(\mathsf{cons}(X, Y)) = X$
   b) $\mathsf{cdr}(\mathsf{cons}(X, Y)) = Y$
   c) $\mathsf{cons}(X, Y) \neq \mathsf{nil}$
2. Axioms for the length
   a) $\mathsf{len}(\mathsf{cons}(X, Y)) = \mathsf{s}(\mathsf{len}(Y))$
   b) $\mathsf{len}(\mathsf{nil}) = 0$
3. Schematic literals of sort ELEM
   a) $\mathsf{car}(a) = e \parallel const(a, e)$
   b) $e_1 = e_2 \parallel const(e_1, e_2)$
   c) $e_1 \neq e_2 \parallel const(e_1, e_2)$
4. Schematic literals of sort LISTS
   a) $\mathsf{cons}(e, a) = b \parallel const(e, a, b)$
   b) $\mathsf{cdr}(a) = b \parallel const(a, b)$
   c) $a = b \parallel const(a, b)$
   d) $a \neq b \parallel const(a, b)$

5. Schematic literals of sort INT
   a) $\mathsf{len}(a) = \mathsf{s}^+(i) \parallel const(a, i)$
   b) $\mathsf{len}(a) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$
   c) $i = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(a, i)$
   d) $i_1 = \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
   e) $i = \mathsf{len}(a) \parallel const(a, i)$
   f) $i_1 = i_2 \parallel const(i_1, i_2)$
   g) $i_1 \neq i_2 \parallel const(i_1, i_2)$

The saturation of $Ax(LLI) \cup G_0$ by $\mathcal{SUPC}_I$ consists of $Ax(LLI) \cup G_0$ and the following schematic literals:

1. $\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
2. $i_1 \neq \mathsf{s}^+(i_1) \parallel const(i_1, i_2)$
3. $\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
4. $\mathsf{s}^+(i_1) = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(i_1, a)$
5. $\mathsf{len}(a) = \mathsf{len}(b) \parallel const(a, b)$
6. $\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$

Indeed, our tool computes the following trace:

$\mathbf{sup}(label(5.d), label(5.d),\ i_1,\ i_1,\ [])$ **gives** $\mathsf{s}^+(i_1) = \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(5.g), label(5.d),\ i_1,\ i_1,\ [])$ **gives** $i_1 \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(2), label(5.d),\ i_1,\ i_1,\ [])$ **gives** $\mathsf{s}^+(i_1) \neq \mathsf{s}^+(i_2) \parallel const(i_1, i_2)$
$\mathbf{sup}(label(5.a), label(5.b),\ \mathsf{len}(a),\ \mathsf{len}(a),\ [])$ **gives** $\mathsf{s}^+(i_1) = \mathsf{s}^+(\mathsf{len}(a)) \parallel const(i_1, a)$
$\mathbf{sup}(label(5.b), label(5.b),\ \mathsf{s}^+(\mathsf{len}(b)),\ \mathsf{s}^+(\mathsf{len}(b)),\ [])$ **gives** $\mathsf{len}(a) = \mathsf{len}(b) \parallel const(a, b)$
$\mathbf{sup}(label(5.b), label(5.b),\ \mathsf{len}(a),\ \mathsf{len}(a),\ [])$ **gives** $\mathsf{s}^+(\mathsf{len}(a)) = \mathsf{s}^+(\mathsf{len}(b)) \parallel const(a, b)$

5.2 Example 2

Consider $Ax(RII) \cup G_0$ that consists of the empty clause and the following literals, where i, j are any integers in $\{1, 2, 3\}$ such that $i \neq j$:

1. Axioms for records
   a) $\mathsf{rselect}_i(\mathsf{rstore}_i(X, Y)) = Y$
   b) $\mathsf{rselect}_j(\mathsf{rstore}_i(X, Y)) = \mathsf{rselect}_j(X, Y)$
2. Axiom for the increment
   a) $\mathsf{rselect}_i(\mathsf{incr}(X)) = \mathsf{s}(\mathsf{rselect}_i(X))$
3. Schematic literals of sort REC
   a) $b = \mathsf{rstore}_i(a, e) \parallel const(a, b, e)$
   b) $b = \mathsf{incr}(a) \parallel const(a, b)$
   c) $a = b \parallel const(a, b)$
   d) $a \neq b \parallel const(a, b)$
4. Schematic literals of sort INT
   a) $e = \mathsf{rselect}_i(a) \parallel const(a, e)$
   b) $\mathsf{rselect}_i(a) = \mathsf{s}^+(e) \parallel const(a, e)$
   c) $\mathsf{rselect}_i(a) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a, b)$
   d) $e = \mathsf{s}^+(\mathsf{rselect}_i(a)) \| const(a, e)$
   e) $e_1 = \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
   f) $e_1 = e_2 \parallel const(e_1, e_2)$
   g) $e_1 \neq e_2 \parallel const(e_1, e_2)$

The saturation of $Ax(RII) \cup G_0$ by $\mathcal{SUPC}_I$ consists of $Ax(RII) \cup G_0$ and the following schematic literals, where i is any integer in $\{1, 2, 3\}$:

1. $\mathsf{s}^+(e_1) = \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
2. $e_1 \neq \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
3. $\mathsf{s}^+(e_1) \neq \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
4. $\mathsf{rselect}_i(a) = \mathsf{rselect}_i(b) \parallel const(a, b)$
5. $\mathsf{s}^+(\mathsf{rselect}_i(a)) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a, b)$
6. $\mathsf{s}^+(e_1) = \mathsf{s}^+(\mathsf{rselect}_i(a)) \parallel const(a, e_1)$
7. $\mathsf{rstore}_i(a, \mathsf{s}^+(e)) = b \parallel const(a, b, e)$

Indeed, our tool computes the following trace:

$\mathbf{sup}(label(4.e), label(4.e),\ e_1,\ e_1,\ [])\ \mathbf{gives}\ \mathsf{s}^+(e_1) = \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
$\mathbf{sup}(label(4.g), label(4.e),\ e_1,\ e_1,\ [])\ \mathbf{gives}\ e_1 \neq \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
$\mathbf{sup}(label(2), label(4.e),\ e_1,\ e_1,\ [])\ \mathbf{gives}\ \mathsf{s}^+(e_1) \neq \mathsf{s}^+(e_2) \parallel const(e_1, e_2)$
$\mathbf{sup}(label(4.c), label(4.c),\ \mathsf{s}^+(\mathsf{rselect}_i(b)),\ \mathsf{s}^+(\mathsf{rselect}_i(b)),\ [])\ \mathbf{gives}$
$\mathsf{rselect}_i(a) = \mathsf{rselect}_i(b) \parallel const(a, b)$
$\mathbf{sup}(label(4.c), label(4.c),\ \mathsf{rselect}_i(a),\ \mathsf{rselect}_i(a),\ [])\ \mathbf{gives}$
$\mathsf{s}^+(\mathsf{rselect}_i(a)) = \mathsf{s}^+(\mathsf{rselect}_i(b)) \parallel const(a, b)$
$\mathbf{sup}(label(4.c), label(4.b),\ \mathsf{rselect}_i(a),\ \mathsf{rselect}_i(a),\ [])\ \mathbf{gives}\ \mathsf{s}^+(e) = \mathsf{s}^+(\mathsf{rselect}_i(a)) \parallel const(a, e)$
$\mathbf{sup}(label(3.a), label(4.e),\ e_1,\ \mathsf{rstore}_i(a, e),\ \mathsf{rstore}_i(a, []),)\ \mathbf{gives}$
$\mathsf{rstore}_i(a, \mathsf{s}^+(e)) = b \parallel const(a, b, e)$

Both examples satisfy Assumption 1 given in Section 3.2. In fact, s-rooted terms occur only in the set of axioms of theories LLI and RII. In both cases the s-rooted term is not the maximal one, therefore, the *Superposition* rule cannot be applied between this axiom and any other literal containing an s^+-rooted term. Let us check that Assumption 2 given in Section 3.3 is also satisfied. Assume m and n are distinct strictly positive integers. Clearly, $\mathsf{s}^m(j)$ and $\mathsf{s}^n(j)$ denote different values, and so we cannot have both $i = \mathsf{s}^m(j)$ and $i = \mathsf{s}^n(j)$ in a saturation of a satisfiable input. Moreover, $\mathsf{rstore}_i(a, \mathsf{s}^m(e))$ and $\mathsf{rstore}_i(a, \mathsf{s}^n(e))$ denote different records, and so we cannot have both $b = \mathsf{rstore}_i(a, \mathsf{s}^m(e))$ and $b = \mathsf{rstore}_i(a, \mathsf{s}^n(e))$ in a saturation of a satisfiable input. Consequently, according to Theorem 2, we can conclude that the paramodulation calculus terminates for both examples.

6 Conclusion

This paper has introduced a new schematic calculus integrating the axioms of the Integer Offsets theory into a framework based on schematic paramodulation. In this context, introducing the s^+ operator together with rewriting rules for terms containing s^+ fits well with automatic verification needs. Indeed, similar abstractions have been successfully used to verify cryptographic protocols with algebraic properties [4], and to prove properties of Java Bytecode programs [3]. Moreover, like in [3], our schematization can be used for fine-tuning the precision of the analysis.

In the present paper the calculus with a new form of schematization for arithmetic expressions has been used to automatically prove the termination of paramodulation modulo Integer Offsets for data structures equipped with counting operators. Our schematic calculus is described as a rule-based system and implemented and validated in the Maude environment.

This paper is the first extension of the notion of schematic paramodulation dedicated to a paramodulation calculus modulo a built-in theory. This study has led to new automatic proof techniques that are different from those performed manually in [14]. The assumptions we use to apply our proof techniques are easy to satisfy for equational theories of practical interest. As future work, we plan to extend this current framework to theories defined by arbitrary clauses, in order to allow for instance arrays with counting operators. Decision procedures for some extensions of the theory of arrays already exist (see, e.g. [5, 9]) but our approach would additionally provide automated proofs of decidability. In this direction, we would have to find a less restrictive assumption to guarantee termination, possibly via a criterion involving the simplification ordering. As in [12], we plan to study some other combinability conditions, and to work on the possibility of determining an upper bound on the number of clauses generated in saturation. Another research direction is to consider a schematic calculus for a more expressive built-in theory of arithmetic, like the theory of Abelian Groups [10, 13].

References

1 A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz. New results on rewrite-based satisfiability procedures. *ACM Trans. Comput. Logic*, 10(1):1 – 51, 2009.

2 A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *J. Inf. Comput*, 183(2):140 – 164, 2003.

3 Y. Boichut, T. Genet, T. P. Jensen, and L. Le Roux. Rewriting approximations for fast prototyping of static analyzers. In F. Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 48–62. Springer, 2007.

4 Y. Boichut, P.-C. Héam, and O. Kouchnarenko. Handling algebraic properties in automatic analysis of security protocols. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *LNCS*, pages 153–167. Springer, 2006.

5 A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

6 M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in Maude 2.4. In R. Treinen, editor, *RTA*, volume 5595 of *LNCS*, pages 380–390. Springer, 2009.

7 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *RTA*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.

8 N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier and MIT Press, 1990.

9 S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.

10 G. Godoy and R. Nieuwenhuis. Superposition with completely built-in abelian groups. *Journal of Symbolic Computation*, 37(1):1–33, 2004.

11 C. Lynch and B. Morawska. Automatic decidability. In *LICS*, pages 7–16, Copenhagen, Denmark, July 2002. IEEE Computer Society.

12 C. Lynch, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic decidability and combinability. *J. Inf. Comput*, 209(7):1026–1047, 2011.

13 E. Nicolini, C. Ringeissen, and M. Rusinowitch. Combinable extensions of Abelian groups. In R. Schmidt, editor, *CADE*, volume 5663 of *LNCS*, pages 51–66. Springer, 2009.

14 E. Nicolini, C. Ringeissen, and M. Rusinowitch. Satisfiability procedures for combination of theories sharing integer offsets. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *LNCS*, pages 428–442. Springer, 2009.

15 R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.

16 E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based framework for building superposition-based decision procedures. In F. Durán, editor, *WRLA*, volume 7571 of *LNCS*, pages 221–239. Springer, 2012.

17 E. Tushkanova, C. Ringeissen, A. Giorgetti, and O. Kouchnarenko. Automatic Decidability for Theories Modulo Integer Offsets. Research Report RR-8139, INRIA, November 2012.

Normalized Completion Revisited*

Sarah Winkler and Aart Middeldorp

Institute of Computer Science
University of Innsbruck, Austria
{sarah.winkler, aart.middeldorp}@uibk.ac.at

Abstract

Normalized completion (Marché 1996) is a widely applicable and efficient technique for completion modulo theories. If successful, a normalized completion procedure computes a rewrite system that allows to decide the validity problem using normalized rewriting. In this paper we consider a slightly simplified inference system for finite normalized completion runs. We prove correctness, show faithfulness of critical pair criteria in our setting, and propose a different notion of normalizing pairs. We then show how normalized completion procedures can benefit from AC-termination tools instead of relying on a fixed AC-compatible reduction order. We outline our implementation of this approach in the completion tool mkbtt and present experimental results, including new completions.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases term rewriting, completion

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.319

1 Introduction

Since the landmark paper of Knuth and Bendix [12], completion has evolved as a basic deduction method in theorem proving, computer algebra and computational logic. Various generalizations have been proposed to deal efficiently with common algebraic theories. The theory of associativity and commutativity (AC) has been incorporated in [16, 22]. For general theories $\mathcal{T}$ where $\mathcal{T}$-unification is finitary and the subterm ordering modulo $\mathcal{T}$ is well-founded, extensions have been presented in [10, 5]. These limitations on the theory have been partially overcome by constrained completion [11], which allows, e.g., for completion modulo AC with a unit element, but excludes other theories such as abelian groups.

Normalized completion [18, 19] constitutes the last result in this line of research. It has three advantages over earlier methods. (1) It allows completion modulo any theory $\mathcal{T}$ that can be represented as an AC-convergent rewrite system $\mathcal{S}$. (2) Critical pairs need not be computed for the theory $\mathcal{T}$, which may not be finitary or even have a decidable unification problem. Instead, any theory between AC and $\mathcal{T}$ can be used. (3) The AC-compatible reduction order used to establish termination need not be compatible with $\mathcal{T}$. This is beneficial for theories such as AC with a unit element where no $\mathcal{T}$-compatible reduction order can possibly exist.

Normalized completion is thus applicable to many common theories such as AC augmented with axioms for unit elements, idempotency or nilpotency, but also to groups and rings. It moreover generalizes Buchberger's algorithm for computing Gröbner bases [20]. Compared to earlier completion techniques, it improves efficiency if the input theory includes a subtheory

* The research described in this paper is supported by a DOC-fFORTE grant of the Austrian Academy of Sciences and the Austrian Science Fund project I963.

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 319–334

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for which an AC-convergent presentation is known. In computing less critical pairs by focusing on a particular theory, the approach shares advantages with efficient specialized theorem proving techniques with built-in equational theories (e.g. [8, 21]).

The focus of this paper is to transform a given theory into a convergent system, in order to obtain a decision procedure which also allows to (dis)prove equational consequences. In this paper we consider a different proof order for finite normalized completion resulting in a slightly simplified inference system. We incorporate critical pair criteria to limit equational consequences, which has been identified as an issue for future work in [17]. The techniques used to obtain fairness, correctness, and completeness results are similar to the ones for standard completion [6], but the setting of normalized completion involves some subtleties. In contrast to [19], we thus make all AC-steps explicit to enhance clarity. Due to some ambiguities concerning the original definition, we also propose a new notion of normalizing pairs which constitute a key ingredient in normalized completion.

State-of-the-art implementations of normalized completion such as C*i*ME require the input of a suitable AC-compatible reduction order. This parameter is critical for success, but hard to determine in advance. We tackle this problem by applying the by now well-understood combination of two approaches: (1) termination tools replace fixed reduction orders as proposed in [25], and (2) back-tracking is avoided by keeping different orientations of equations. This combined multi-completion approach with termination tools has been investigated for standard completion [29], ordered completion [27] and AC-completion [28]. We present novel convergent systems obtained with our method.

The remainder of this paper is structured as follows. Preliminaries on equational reasoning and rewriting are given in Section 2. In Section 3 we recall normalized completion, present correctness and completeness result based on our proof order, and describe critical pair criteria in the setting of normalized completion. Section 4 describes the extension with termination tools. In Section 5 we give a short description of our implementation in mkbtt, outline the multi-completion approach and some implementation details, and present experimental results. In Section 6 we conclude. Due to a lack of space, some (proof) details can be found in the first author's PhD thesis [26, Chapter 6].

2 Preliminaries

We assume familiarity with term rewriting and Knuth-Bendix completion [3], and recall only some central notions. We consider term rewrite systems (TRSs) $\mathcal{R}$ over a signature $\mathcal{F}$. If the associated rewrite relation $\to_{\mathcal{R}}$ is well-founded, we write $s \to^!_{\mathcal{R}} t$ if s rewrites to a normal form t, and $s{\downarrow_{\mathcal{R}}}$ to denote some $\mathcal{R}$-normal form of s. We also consider (symmetric) equational systems $\mathcal{E}$ over $\mathcal{F}$ with associated equational theory $=_{\mathcal{E}}$. If $u \approx v$ is an equation in $\mathcal{E}$ we write $u \simeq v$ to denote $u \approx v$ or $v \approx u$. Let $\mathcal{F}_{\mathrm{AC}} \subseteq \mathcal{F}$ be a set of binary function symbols. The equational system AC contains equations $x + (y + z) \approx (x + y) + z$ and $x + y \approx y + x$ for all symbols $+ \in \mathcal{F}_{\mathrm{AC}}$. We denote equivalence modulo AC by $\leftrightarrow^*_{\mathrm{AC}}$. A term s *rewrites* to t in $\mathcal{R}$ *modulo AC*, denoted by $s \to_{\mathcal{R}/\mathrm{AC}} t$, whenever $s \leftrightarrow^*_{\mathrm{AC}} \cdot \to_{\mathcal{R}} \cdot \leftrightarrow^*_{\mathrm{AC}} t$ holds.

A TRS $\mathcal{R}$ *terminates modulo* AC whenever the relation $\to_{\mathcal{R}/\mathrm{AC}}$ is well-founded. To establish AC-termination we will consider AC-compatible reduction orders $\succ$, i.e., reduction orders that satisfy $\leftrightarrow^*_{\mathrm{AC}} \cdot \succ \cdot \leftrightarrow^*_{\mathrm{AC}} \subseteq \succ$. The TRS $\mathcal{R}$ is *convergent modulo AC* if it terminates modulo AC and the relation $\leftrightarrow^*_{\mathrm{AC} \cup \mathcal{R}}$ coincides with $\to^*_{\mathcal{R}/\mathrm{AC}} \cdot \leftrightarrow^*_{\mathrm{AC}} \cdot \leftarrow^*_{\mathcal{R}/\mathrm{AC}}$.

Let $\mathcal{L}$ be a theory with finitary and decidable unification problem. A substitution σ constitutes an $\mathcal{L}$*-unifier* of two terms s and t if $s\sigma \leftrightarrow^*_{\mathcal{L}} t\sigma$ holds. An $\mathcal{L}$*-overlap* is a quadruple $\langle \ell_1 \to r_1, p, \ell_2 \to r_2 \rangle_\Sigma$ consisting of rewrite rules $\ell_1 \to r_1$, $\ell_2 \to r_2$, a position $p \in \mathcal{P}\mathrm{os}_{\mathcal{F}}(\ell_2)$,

and a complete set Σ of $\mathcal{L}$-unifiers of $\ell_2|_p$ and ℓ_1. Then $\ell_2\sigma[r_1\sigma]_p \leftarrow \rtimes \rightarrow r_2\sigma$ constitutes an $\mathcal{L}$-*critical pair* for every $\sigma \in \Sigma$. We write $s \leftarrow \times \rightarrow t$ if $s \leftarrow \rtimes \rightarrow t$ or $t \leftarrow \rtimes \rightarrow s$ is an $\mathcal{L}$-critical pair. For two sets of rewrite rules $\mathcal{R}_1$ and $\mathcal{R}_2$, we also write $\mathrm{CP}_{\mathcal{L}}(\mathcal{R}_1, \mathcal{R}_2)$ for the set of all $\mathcal{L}$-critical pairs emerging from an overlap where $\ell_1 \to r_1 \in \mathcal{R}_1$ and $\ell_2 \to r_2 \in \mathcal{R}_2$, and $\mathrm{CP}_{\mathcal{L}}(\mathcal{R}_1)$ for the set of all $\mathcal{L}$-critical pairs such that $\ell_1 \to r_1$, $\ell_2 \to r_2 \in \mathcal{R}_1$. A peak $s \,{}^{p}_{r\leftarrow\ell}\!\leftarrow \cdot \leftrightarrow^*_{\mathcal{T}} \cdot \to^{q}_{u\to v} t$ is called a *non-overlap* if it is not an instance of an $\mathcal{L}$-overlap.

For a rewrite rule $\ell \to r$ with $+ \in \mathcal{F}_{\mathrm{AC}}$ we write $(\ell \to r)^e$ for the *extended rule* $\ell + x \to r + x$, where $x \in \mathcal{V}$ is fresh. The TRS $\mathcal{R}^e$ contains all rules in $\mathcal{R}$ plus all extended rules $\ell + x \to r + x$ such that $\ell \to r \in \mathcal{R}$ [4].

In normalized completion, we consider a fixed rewrite system $\mathcal{S}$ and a pair $(\mathcal{E}, \mathcal{R})$ of equations $\mathcal{E}$ and rewrite rules $\mathcal{R}$. An *equational proof step* $s \leftrightarrow^{p,\sigma}_{e} t$ in $(\mathcal{S}, \mathcal{E}, \mathcal{R})$ is an *AC-step* (*equality step*) if e or e^{-1} is an equation in AC ($\mathcal{E}$) applied from left to right at position p in s with substitution σ. A proof step $s \leftrightarrow^{p,\sigma}_{\ell\to r} t$ is a *rewrite step* if $s = u[\ell\sigma]_p$ and $t = u[r\sigma]_p$ for some term u with position p and substitution σ and rewrite rule $\ell \to r$ in $\mathcal{R}$ or $\mathcal{S}$. In this case also $t \leftrightarrow^{p,\sigma}_{r\leftarrow\ell} s$ is a rewrite proof step. We call a proof step an $\mathcal{R}$-rewrite ($\mathcal{S}$-rewrite) step if it is a rewrite step using a rule in $\mathcal{R}$ ($\mathcal{S}$).

We sometimes write $s \leftrightarrow t$ to express the existence of some proof step, omitting the position p, substitution σ and equation or rule e. An *equational proof* P of an equation $t_0 \approx t_n$ is a finite sequence

$$t_0 \underset{e_0}{\overset{p_0}{\leftrightarrow}} t_1 \underset{e_1}{\overset{p_1}{\leftrightarrow}} \cdots \underset{e_{n-1}}{\overset{p_{n-1}}{\leftrightarrow}} t_n \tag{1}$$

of equational proof steps. It has a *subproof* Q, denoted by $P[Q]$, if Q is a sequence $t_i \leftrightarrow \cdots \leftrightarrow t_j$ with $0 \leqslant i \leqslant j \leqslant n$. For a term u with position q, a substitution σ, and a proof P of the shape (1) we write $u[P\sigma]_q$ to denote the sequence

$$u[t_0\sigma]_q \underset{e_0}{\overset{qp_0}{\leftrightarrow}} u[t_1\sigma]_q \underset{e_1}{\overset{qp_1}{\leftrightarrow}} \cdots \underset{e_{n-1}}{\overset{qp_{n-1}}{\leftrightarrow}} u[t_n\sigma]_q$$

which is again an equational proof. A *proof order* $\succ$ is a well-founded order on equational proofs such that (1) $P \succ Q$ implies $u[P\sigma]_p \succ u[Q\sigma]_p$ for all substitutions σ and terms u with position p, and (2) $P \succ P'$ implies $Q[P] \succ Q[P']$ for all proofs P, P' and Q.

In the sequel we will consider a fixed theory $\mathcal{T}$ that is representable as an AC-convergent rewrite system $\mathcal{S}$,[1] so $\leftrightarrow^*_{\mathcal{T}} = \to^!_{\mathcal{S}/\mathrm{AC}} \cdot \leftrightarrow^*_{\mathrm{AC}} \cdot \leftarrow^!_{\mathcal{S}/\mathrm{AC}}$. For example, for the theory ACU consisting of an AC-operator $+$ with unit $\mathsf{0}$, we have $\mathcal{T} = \{x + (y + z) \approx (x + y) + z,\ x + y \approx y + x,\ x + \mathsf{0} \approx x\}$ and $\mathcal{S} = \{x + \mathsf{0} \to x\}$. Note that the representation $\mathcal{S}$ need not be unique.

We define normalized rewriting as in [19] but use a different notation to distinguish it from the by now established notation for rewriting modulo. Two terms s and t admit an $\mathcal{S}$-*normalized* $\mathcal{R}$-*rewrite step* if

$$s \underset{\mathcal{S}/\mathrm{AC}}{\overset{!}{\longrightarrow}} s' \underset{\mathrm{AC}}{\overset{*}{\leftrightarrow}} \cdot \underset{\ell\to r}{\overset{p}{\longrightarrow}} \cdot \underset{\mathrm{AC}}{\overset{*}{\leftrightarrow}} t \tag{2}$$

for some rule $\ell \to r$ in $\mathcal{R}$ and position p in s'. We write $s \to^p_{\ell\to r\setminus\mathcal{S}} t$ for (2), and $s \to_{\mathcal{R}\setminus\mathcal{S}} t$ if $s \to^p_{\ell\to r\setminus\mathcal{S}} t$ for a rule $\ell \to r$ in $\mathcal{R}$ and position p.

Let $\succ$ be an AC-compatible reduction order such that $\mathcal{S} \subseteq \succ$. For any set of rewrite rules $\mathcal{R}$ satisfying $\mathcal{R} \subseteq \succ$ the normalized rewrite relation $\to_{\mathcal{R}\setminus\mathcal{S}}$ is well-founded [18, 19], so we

[1] To avoid confusion we differentiate between the theory and its AC-convergent representation, although both are denoted by S in [19].

deduce	$\dfrac{\mathcal{E},\mathcal{R}}{\mathcal{E}\cup\{s\approx t\},\mathcal{R}}$	if $s\approx t\in \mathrm{CP}_{\mathcal{L}}(\mathcal{R})$	simplify	$\dfrac{\mathcal{E}\cup\{s\simeq t\},\mathcal{R}}{\mathcal{E}\cup\{s\simeq u\},\mathcal{R}}$	if $t\to_{\mathcal{R}\setminus\mathcal{S}} u$
normalize	$\dfrac{\mathcal{E}\cup\{s\approx t\},\mathcal{R}}{\mathcal{E}\cup\{s{\downarrow}\approx t{\downarrow}\},\mathcal{R}}$	if $s\neq s{\downarrow}$ or $t\neq t{\downarrow}$	compose	$\dfrac{\mathcal{E},\mathcal{R}\cup\{s\to t\}}{\mathcal{E},\mathcal{R}\cup\{s\to u\}}$	if $t\to_{\mathcal{R}\setminus\mathcal{S}} u$
delete	$\dfrac{\mathcal{E}\cup\{s\approx t\},\mathcal{R}}{\mathcal{E},\mathcal{R}}$	if $s\leftrightarrow^*_{\mathrm{AC}} t$	collapse	$\dfrac{\mathcal{E},\mathcal{R}\cup\{t\to s\}}{\mathcal{E}\cup\{u\approx s\},\mathcal{R}}$	if $t\to_{\mathcal{R}\setminus\mathcal{S}} u$
orient	$\dfrac{\mathcal{E}\cup\{s\simeq t\},\mathcal{R}}{\mathcal{E}\cup\Theta(s,t),\mathcal{R}\cup\Psi(s,t)}$	if $s=s{\downarrow}$ and $t=t{\downarrow}$			

Figure 1 $\mathcal{S}$-normalized completion NKB.

can consider equational proofs of the form $s \xrightarrow[\mathcal{R}\setminus\mathcal{S}]{!} \cdot \xleftrightarrow[\mathcal{T}]{*} \cdot \xleftarrow[\mathcal{R}\setminus\mathcal{S}]{!} t$. These normal form proofs play a special role and are called *normalized rewrite proofs*. Because $\mathcal{S}$ is AC-convergent for $\mathcal{T}$, any such proof can be transformed into a proof $s \downdownarrows_{\mathcal{R}\setminus\mathcal{S}} t$, where $\downdownarrows_{\mathcal{R}\setminus\mathcal{S}}$ abbreviates the relation $\xrightarrow[\mathcal{R}\setminus\mathcal{S}]{!} \cdot \xrightarrow[\mathcal{S}/\mathrm{AC}]{!} \cdot \xleftrightarrow[\mathrm{AC}]{*} \cdot \xleftarrow[\mathcal{S}/\mathrm{AC}]{!} \cdot \xleftarrow[\mathcal{R}\setminus\mathcal{S}]{!}$. A TRS $\mathcal{R}$ is called *$\mathcal{S}$-convergent for* a set of equations $\mathcal{E}$ if $\to_{\mathcal{R}\setminus\mathcal{S}}$ is terminating and the relations $\leftrightarrow^*_{\mathcal{E}\cup\mathcal{T}}$ and $\xrightarrow[\mathcal{R}\setminus\mathcal{S}]{!} \cdot \xleftrightarrow[\mathcal{T}]{*} \cdot \xleftarrow[\mathcal{R}\setminus\mathcal{S}]{!}$ coincide.

3 Normalized Completion

Let $\mathcal{S}$ be AC-convergent for $\mathcal{T}$, and $\succ$ be an AC-compatible reduction order such that $\mathcal{S}\subseteq\succ$. From now on we write $t{\downarrow}$ for $t{\downarrow}_{\mathcal{S}/\mathrm{AC}}$ and $s{\downarrow}_p$ for $s[u{\downarrow}]_p$ where $u=s|_p$. We let $c(s,p,t)$ denote the multiset $\{s\}$ if $s{\downarrow}_p=s$ and $\{s,t\}$ otherwise.

Figure 1 displays the inference system NKB. Note that the collapse rule slightly differs from the version in [19] in that no (strict encompassment) restriction is made on the applied rule in $\mathcal{R}$. This simplification is inspired by the similar modification to standard completion presented in [24] and possible because we restrict to *finite* runs. In the deduce rule, $\mathcal{L}$ denotes some fixed theory such that $\mathrm{AC}\subseteq\mathcal{L}\subseteq\mathcal{T}$.[2] The normalize rule replaces terms in an equation by their normal forms with respect to $\mathcal{S}$, provided that at least one term is not $\mathcal{S}$-normalized. This restriction is missing in [19], but required to ensure progress.

In the orient rule, $\Theta(s,t)$ is a set of equations and $\Psi(s,t)$ is a set of rewrite rules. These functions will be chosen in a way such that (Θ,Ψ) forms a *normalizing pair*. Before giving the definition of this crucial ingredient to normalized completion, we define some properties of inference sequences and our proof order. An inference sequence

$$\gamma\colon (\mathcal{E}_0,\varnothing)\vdash(\mathcal{E}_1,\mathcal{R}_1)\vdash(\mathcal{E}_2,\mathcal{R}_2)\vdash\cdots\vdash(\mathcal{E}_n,\mathcal{R}_n) \tag{3}$$

using the rules in Figure 1 is called a *run* of length n. Throughout this paper we will restrict to *finite* runs,[3] and denote the n-fold composition of the inference relation by $\vdash^n$. A run *fails* if $\mathcal{E}_n$ is non-empty, it *succeeds* if $\mathcal{E}_n$ is empty and $\mathcal{R}_n$ is $\mathcal{S}$-convergent for $\mathcal{E}_0$. We assume that all equations and rewrite rules appearing in a run are variable-disjoint, i.e., any equation or

[2] Thus if $\mathcal{T}$ itself is not decidable and finitary with respect to unification, one can simply use AC for $\mathcal{L}$. On the other hand, for example the set of unifiers obtained from ACU or ACUI unification are typically much smaller than those obtained from AC unification.

[3] Normalized completion for infinite runs is discussed in [26].

rule added in a step $(\mathcal{E}_k, \mathcal{R}_k) \vdash (\mathcal{E}_{k+1}, \mathcal{R}_{k+1})$ is variable-disjoint from $\bigcup_{1 \leqslant i \leqslant k} \mathcal{E}_i \cup \mathcal{R}_i$. We now define a proof reduction relation $\Rightarrow_n$ that depends on the actual run under consideration.

▶ **Definition 3.1.** Consider a run (3) of length n using the reduction order $\succ$, and some $(\mathcal{E}_i, \mathcal{R}_i)$ for $0 \leqslant i \leqslant n$. The *cost* c_n of a proof step in $(\mathcal{T}, \mathcal{E}_i, \mathcal{R}_i)$ is defined as follows:

$$
\begin{aligned}
c_n(s \xleftrightarrow[u \approx v]{} t) &= (\bot, \{s\}, \bot, 0) && \text{if } u \simeq v \in \mathrm{AC} \\
c_n(s \xleftrightarrow[u \approx v]{p} t) &= (\{s\!\downarrow_p, t\!\downarrow_p\}, \{s, t\}, \bot, 0) && \text{if } u \simeq v \in \mathcal{E}_i \\
c_n(s \xleftrightarrow[\ell \to r]{p} t) &= c_n(t \xleftrightarrow[r \leftarrow \ell]{p} s) = (c(s,p,t), \{s\}, (s|_p)\!\downarrow, n-k) && \text{if } k \text{ is maximal such that} \\
& && \ell \to r \in \mathcal{R}_k \\
c_n(s \xleftrightarrow[\ell \to r]{} t) &= c_n(t \xleftrightarrow[r \leftarrow \ell]{} s) = (\bot, \{s\}, \bot, 0) && \text{if } \ell \to r \in \mathcal{S}
\end{aligned}
$$

We compare costs with the lexicographic combination of $(\succ_{\mathsf{mul}}, (\leftrightarrow^*_{\mathrm{AC}})_{\mathsf{mul}})$ for the first two components, $(\rhd_{\mathrm{AC}}, \leftrightarrow^*_{\mathrm{AC}})$, and $(>, =)$ for the standard order $>$ on $\mathbb{N}$. The symbol $\bot$ is considered minimal in the former three orderings. The cost of an equational proof is the multiset consisting of the costs of its steps. The proof order $\succapprox_n$ is the multiset extension of the order on proof step costs, and $P \Rightarrow_n Q$ holds if and only if $P \succapprox_n Q$ and P and Q prove the same equation.

As the multiset extension of a lexicographic combination of well-founded orders, the relation $\succapprox_n$ is well-founded. Hence the following is not difficult to show.

▶ **Lemma 3.2.** *The relation $\Rightarrow_n$ is a proof reduction relation.* ◀

It is easy to see that NKB is sound in that the equational theory is not modified.

▶ **Soundness Lemma 3.3.** *In any run (3) the relations $\leftrightarrow^*_{\mathcal{E}_0 \cup \mathcal{T}}$ and $\leftrightarrow^*_{\mathcal{E}_n \cup \mathcal{R}_n \cup \mathcal{T}}$ coincide.* ◀

The following lemma links the proof reduction relation $\Rightarrow_n$ to our inference system NKB.

▶ **Persistence Lemma 3.4.** *Consider a run of the form (3) and let P be an equational proof in $(\mathcal{S}, \mathcal{E}_i, \mathcal{R}_i)$ for $1 \leqslant i \leqslant n$. Then there is a proof Q in $(\mathcal{S}, \mathcal{E}_n, \mathcal{R}_n)$ such that $P \Rightarrow_n^= Q$.* ◀

We next state an AC version of the Extended Critical Pair Lemma [9, 10].

▶ **Lemma 3.5.** *Let $\ell_1 \to r_1$ and $\ell_2 \to r_2$ admit a peak $P\colon s \,{}_{r_1 \leftarrow \ell_1}\!\leftarrow \cdot \leftrightarrow^*_{\mathrm{AC}} \cdot \to_{\ell_2 \to r_2} t$. If P does not contain an instance of an AC overlap then $s \to^*_{\ell_2 \to r_2/\mathrm{AC}} \cdot \,{}^*_{r_1 \leftarrow \ell_1/\mathrm{AC}}\!\leftarrow t$. Otherwise, there is a critical pair $u \leftarrow \times \rightarrow v$ in $\mathrm{CP}_{\mathrm{AC}}(\ell_1 \to r_1, \ell_2 \to r_2)$ or $\mathrm{CP}_{\mathrm{AC}}(\ell_1 \to r_1, (\ell_2 \to r_2)^e)$ such that $s \xleftrightarrow[\mathrm{AC}]{*} \cdot \xleftrightarrow[u \approx v]{} \cdot \xleftrightarrow[\mathrm{AC}]{*} t$.* ◀

Note that this implies that any non-joinable peak is an instance of an AC-critical pair between two rules where *at most one* rule is extended, so critical pairs between two extended rules of a rewrite system $\mathcal{R}$ can be ignored. Moreover, it suffices to extend one rule, no matter which one. The following lemma builds upon the previous statement and shows that both joining sequences and critical pairs admit smaller proofs. A proof can be found in [26].

▶ **Lemma 3.6.** *Let $\mathcal{R}$ be a set of rewrite rules such that $\mathcal{R} \subseteq \succ$ and let $n \geqslant 0$.*

(a) *If $P\colon s \,{}_{\mathcal{S}}\!\leftarrow u \leftrightarrow^*_{\mathrm{AC}} u' \to_{\mathcal{S}} t$ then $P \Rightarrow_n Q$ for some proof $Q\colon s \to^*_{\mathcal{S}/\mathrm{AC}} \cdot \,{}^*_{\mathcal{S}/\mathrm{AC}}\!\leftarrow t$.*

(b) *If $P\colon s \,{}_{\mathcal{R}}\!\leftarrow u \leftrightarrow^*_{\mathrm{AC}} u' \to_{\mathcal{R}} t$ then we have $Q\colon s \to^*_{\mathcal{R}/\mathrm{AC}} \cdot \,{}^*_{\mathcal{R}/\mathrm{AC}}\!\leftarrow t$ such that $P \Rightarrow_n Q$, or there is some critical pair $s' \leftarrow \times \rightarrow t'$ in $\mathrm{CP}_{\mathrm{AC}}(\mathcal{R}, \mathcal{R}^e)$ such that $P \Rightarrow_n s \leftrightarrow^*_{\mathrm{AC}} \cdot \leftrightarrow_{s' \approx t'} \cdot \leftrightarrow^*_{\mathrm{AC}} t$.*

(c) *If* $P\colon s \mathrel{\mathcal{R}}\!\leftarrow u \leftrightarrow^*_{\mathrm{AC}} u' \to_{\mathcal{S}} t$ *then there is a proof* $Q\colon s \to^*_{\mathcal{S}/\mathrm{AC}} \cdot \mathrel{\mathcal{R}/\mathrm{AC}}^*\!\leftarrow t$ *such that* $P \Rightarrow_n Q$*, or there is a critical pair* $s' \leftarrow \times \to t'$ *in* $\mathrm{CP}_{\mathrm{AC}}(\mathcal{R}, \mathcal{S}^e) \cup \mathrm{CP}_{\mathrm{AC}}(\mathcal{S}, \mathcal{R})$ *such that* $P \Rightarrow_n s \leftrightarrow^*_{\mathrm{AC}} \cdot \leftrightarrow_{s' \approx t'} \cdot \leftrightarrow^*_{\mathrm{AC}} t$. ◀

We are now ready to define the crucial concept of $\mathcal{S}$-normalizing pairs.[4]

▶ **Definition 3.7.** Let $(\mathcal{E}_i, \mathcal{R}_i)$ occur in a run of the form (3) and u, v be terms such that $u \simeq v \in \mathcal{E}_i$. Let furthermore Θ and Ψ be functions such that $\Theta(u, v)$ is a set of equations and $\Psi(u, v)$ is a set of rewrite rules. Then (Θ, Ψ) constitutes an *$\mathcal{S}$-normalizing pair* for u and v if

(i) $\Theta(u, v)$ and $\Psi(u, v)$ are contained in $\leftrightarrow^*_{\mathcal{E}_i \cup \mathcal{R}_i \cup \mathcal{T}}$, and $\Psi(u, v) \subseteq \succ$,

(ii) for every proof P of the shape $s \xleftarrow[u \approx v]{\epsilon, \sigma} t$ there exists a proof Q in $(\mathcal{T}, \Theta(u, v), \Psi(u, v))$ such that $P \Rightarrow_n Q$, and

(iii) for all rules $\ell \to r$ in $\Psi(u, v)$, all sets of rewrite rules $\mathcal{R}$ and all proofs P of the form $s \mathrel{\mathcal{S}}\!\leftarrow w \leftrightarrow^*_{\mathrm{AC}} \cdot \to_{\ell \to r} \cdot \to^*_{\mathcal{R} \setminus \mathcal{S}} t$ there is a proof Q in $(\mathcal{T}, \Theta(u, v), \Psi(u, v) \cup \mathcal{R})$ such that $P \Rightarrow_n Q$, and all terms in Q are smaller than w.

Here condition (i) ensures that soundness and termination are preserved. Condition (ii) requires that all proofs using the equation $u \approx v$ can be replaced by smaller proofs, which is often achieved by adding the rule $u \to v$. Condition (iii) takes AC overlaps between rules in $\Psi(u, v)$ and $\mathcal{S}$ into account, but since rules in $\Psi(u, v)$ may at a later stage get composed with other rules, the considered peaks take a more general shape. In the sequel all orient steps in runs will be assumed to apply $\mathcal{S}$-normalizing pairs.

▶ **Example 3.8.** Take the theory ACU where $\mathcal{S} = \{x + 0 \to x\}$ and consider the $\mathcal{S}$-normalized terms $u = -(x + y)$ and $v = (-x) + (-y)$. Let $\succ$ be an AC-RPO. If the precedence is $- \succ + \succ 0$, we have $u \succ v$. Then $\Theta(u, v) = \{-x \approx (-x) + (-0)\}$ and $\Psi(u, v) = \{u \to v\}$ form a valid normalizing pair:[5] Condition (i) is clearly satisfied. Condition (ii) holds as any proof using $u \approx v$ can be transformed into a proof using $u \to v$ which is smaller by Definition 3.1 as $u = u{\downarrow}$. Finally, using Lemma 3.6 it is not hard to see that by adding the AC-critical pair in $\Theta(u, v)$ also condition (iii) holds. If the precedence is $+ \succ - \succ 0$ such that $v \succ u$, one may simply take $\Theta(v, u) = \varnothing$ and $\Psi(v, u) = \{v \to u\}$.

Marché proposes a *general $\mathcal{S}$-normalizing pair* which is applicable for any choice of the theory $\mathcal{S}$, where $\Psi(u, v)$ consists of the oriented term pair $u \to v$ and $\Theta(u, v)$ contains AC-critical pairs between $u \to v$ and a rule in $\mathcal{S}$:

▶ **Definition 3.9** ([19, Definition 3.9])**.** Let u and v be terms in $\mathcal{S}$-normal form such that $u \succ v$. The *general normalizing pair* $(\Theta_{\mathrm{gen}}, \Psi_{\mathrm{gen}})$ is defined by $\Psi_{\mathrm{gen}}(u, v) = \{u \to v\}$ and $\Theta_{\mathrm{gen}}(u, v) = \mathrm{CP}_{\mathrm{AC}}(u \to v, \mathcal{S}^e) \cup \mathrm{CP}_{\mathrm{AC}}(\mathcal{S}, u \to v)$.

We now prove that $(\Theta_{\mathrm{gen}}, \Psi_{\mathrm{gen}})$ is also a normalizing pair according to Definition 3.7.

▶ **Lemma 3.10.** *Let $\mathcal{E}_i$ occur in some run* (3) *and $u \simeq v \in \mathcal{E}_i$. If u and v are terms in $\mathcal{S}$-normal form such that $u \succ v$ then $(\Theta_{\mathrm{gen}}, \Psi_{\mathrm{gen}})$ forms a normalizing pair.*

---

[4] The definition of normalizing pairs varies in the literature; the first reference in [17, Definition 4.4] is different from [19, Definition 3.5] and [20, Definition 3.1]. But none of these definitions allowed us to understand and reproduce the correctness proof (cf. the remarks on page 327), thus we use a different notion.

[5] These functions are instances of ACU-normalizing pairs [19].

Proof. We argue that $\Theta_{\text{gen}}(u,v)$ and $\Psi_{\text{gen}}(u,v)$ satisfy the three requirements demanded in Definition 3.7. Condition (i) is satisfied as due to $u \simeq v \in \mathcal{E}_i$ both $\Theta_{\text{gen}}(u,v)$ and $\Psi_{\text{gen}}(u,v)$ are contained in $\leftrightarrow^*_{\mathcal{E}_i \cup \mathcal{T}}$, and $\Psi_{\text{gen}}(u,v) \subseteq \succ$ as $u \succ v$.

Concerning condition (ii), any proof $P\colon s \leftrightarrow^{\epsilon,\sigma}_{u \approx v} t$ can be transformed into $Q\colon s \leftrightarrow^{\epsilon,\sigma}_{u \to v} t$. We obtain the decrease $u \leftrightarrow^{\epsilon}_{u \approx v} v \Rightarrow_n u \leftrightarrow^{\epsilon}_{u \to v} v$ because $\{(\{u,v\},\ldots)\} \succcurlyeq_n \{(\{u\},\ldots)\}$. As $\Rightarrow_n$ is a proof reduction relation also $P \Rightarrow_n Q$ holds.

Finally, consider a proof P of the form $s \mathrel{\mathcal{S}}\!\leftarrow w \leftrightarrow^*_{\text{AC}} w' \to^p_{u \to v} t \to_{\mathcal{R} \setminus \mathcal{S}} \hat{t}$. By Lemma 3.6, there exists a smaller proof of $s \approx t$ (and thus also of $s \approx \hat{t}$) if the peak in P does not constitute a proper overlap. Otherwise P must contain an instance of an AC-critical peak, so $s \leftrightarrow^*_{\text{AC}} C[s'\sigma]$ and $t \leftrightarrow^*_{\text{AC}} C[t'\sigma]$ for some context C, substitution σ, and AC-critical pair $s' \simeq t'$. According to Lemma 3.5 we may assume that one rule comes from $\mathcal{S}^e$ and one rule comes from $\mathcal{R}$. Hence $s' \simeq t' \in \Theta_{\text{gen}}(u,v)$, which gives rise to the proof Q of the form $s \leftrightarrow^*_{\text{AC}} \cdot \leftrightarrow_{s' \approx t'} \cdot \leftrightarrow^*_{\text{AC}} t \to_{\mathcal{R} \setminus \mathcal{S}} \hat{t}$. We have $c_n(P) = \{(\bot, \{w\}, \ldots), (c(w',p,t),\ldots)\} \cup c_{\text{AC}}(P) \cup c_n(P')$ for $P'\colon t \to_{\mathcal{R} \setminus \mathcal{S}} \hat{t}$, whereas $c_n(Q) = \{(\{C[s'\sigma]{\downarrow}, C[t'\sigma]{\downarrow}\},\ldots)\} \cup c_{\text{AC}}(Q) \cup c_n(P')$. We have $w' \succ t$, and by AC compatibility also $w' \succ s$, $w' \succ C[s'\sigma] \succ C[s'\sigma]{\downarrow}$, and $w' \succ C[t'\sigma] \succ C[t'\sigma]{\downarrow}$. Thus $(c(w',p,t),\ldots) \in c_n(P)$ is greater than all cost tuples in $c_n(Q)$, so $P \Rightarrow_n Q$. This shows that also condition (iii) is satisfied. ◀

3.1 Fairness and Correctness

Fairness captures the important property of runs that whenever some inference step can achieve progress then progress is eventually made.

▶ **Definition 3.11.** A nonfailing NKB run $(\mathcal{E}_0, \varnothing) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \cdots \vdash (\varnothing, \mathcal{R}_n)$ is *fair with respect to* $\Rightarrow_n$ if for any proof P in $\mathcal{T} \cup \mathcal{R}_n$ which is not a rewrite proof there is a proof Q in $(\mathcal{T}, \mathcal{E}_i, \mathcal{R}_i)$ for some $0 \leqslant i \leqslant n$ such that $P \Rightarrow_n Q$.

Note that our definition is less restrictive than the original one, which is essential to incorporate critical pair criteria (see Section 3.2). We show that the original definition [19] constitutes a sufficient criterion for fairness in our sense. Beforehand, we state two technical results about persistent rules and $\mathcal{L}$-critical pairs. Their proofs can be found in [26].

▶ **Lemma 3.12.** *Assume an* NKB *run* $(\mathcal{E}_0, \varnothing) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \cdots \vdash (\mathcal{E}_n, \mathcal{R}_n)$ *has a rule* $\ell \to r \in \mathcal{R}_n$ *giving rise to a peak* $P\colon s \mathrel{\mathcal{S}}\!\leftarrow w \leftrightarrow^*_{\text{AC}} w' \to^p_{\ell \to r \setminus \mathcal{S}} t$. *Then there is a proof* P' *in* $(\mathcal{T}, \mathcal{E}_n, \mathcal{R}_n)$ *such that* $P \Rightarrow_n P'$, *and for all* $(T,\ldots) \in c_n(P')$ *the set* T *contains only terms which are smaller than* w. ◀

▶ **Lemma 3.13.** *Let* $\ell_1 \to r_1$ *and* $\ell_2 \to r_2$ *be rewrite rules and* $\text{AC} \subseteq \mathcal{L} \subseteq \mathcal{T}$. *If* $s \simeq t \in \text{CP}_{\text{AC}}(\ell_1 \to r_1, \ell_2 \to r_2)$ *then there is some critical pair* $s' \leftarrow \times \to t' \in \text{CP}_{\mathcal{L}}(\ell_1 \to r_1, \ell_2 \to r_2)$ *and substitution* ρ *such that* $s \leftrightarrow^*_{\mathcal{T}} s'\rho$ *and* $t \leftrightarrow^*_{\mathcal{T}} t'\rho$. ◀

▶ **Lemma 3.14.** *A nonfailing* NKB *run* $(\mathcal{E}_0, \varnothing) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \cdots \vdash (\varnothing, \mathcal{R}_n)$ *satisfying* $\text{CP}_{\mathcal{L}}(\mathcal{R}_n, \mathcal{R}_n^e) \subseteq \bigcup_i \mathcal{E}_i$ *is fair with respect to* $\Rightarrow_n$.

Proof. We show that every proof in $\mathcal{T} \cup \mathcal{R}_n$ which is minimal with respect to $\Rightarrow_n$ is a normalized rewrite proof. Assume to the contrary that P minimal but not a rewrite proof. Thus P contains (i) a peak $s \mathrel{\mathcal{R}_n}\!\leftarrow \cdot \leftrightarrow^*_{\text{AC}} \cdot \to_{\mathcal{R}_n} t$, or (ii) a peak $s \mathrel{\mathcal{R}_n/\text{AC}}\!\leftarrow \cdot \to_{\mathcal{S}/\text{AC}} t$ or $s \mathrel{\mathcal{S}/\text{AC}}\!\leftarrow \cdot \to_{\mathcal{R}_n/\text{AC}} t$, or (iii) a subproof $u \to_{\mathcal{R}_n/\text{AC}} t$ such that $u \neq u{\downarrow}$, or (iv) a peak $s \mathrel{\mathcal{S}/\text{AC}}\!\leftarrow \cdot \to_{\mathcal{S}/\text{AC}} t$. For each case we show that a smaller proof contradicts minimality of P.

If a peak of the form (i) originates from a non-overlap then by Lemma 3.6(b) it could be replaced by a smaller proof. Otherwise, by Lemma 3.5 the peak $s \mathrel{\mathcal{R}_n}\!\leftarrow \cdot \leftrightarrow^*_{\text{AC}} \cdot \to_{\mathcal{R}_n} t$ must

satisfy $s \leftrightarrow^*_{\mathrm{AC}} C[s'\sigma]$ and $t \leftrightarrow^*_{\mathrm{AC}} C[t'\sigma]$ for some critical pair $s' \leftarrow \times \rightarrow t'$ in $\mathrm{CP}_{\mathrm{AC}}(\mathcal{R}_n, \mathcal{R}_n^e)$. Assume $s' \leftarrow \times \rightarrow t'$ originates from a peak $P'\colon s' \;{}_{\mathcal{R}_n}{}^{p}\!\leftarrow w \leftrightarrow^*_{\mathrm{AC}} w' \rightarrow^q_{\mathcal{R}_n^e} t'$. We show that $\mathcal{T} \cup \mathcal{R}_n$ admits a smaller proof than P', which entails the existence of a smaller proof than P. By Lemma 3.13 there must also be an $\mathcal{L}$-critical pair $s'' \approx t''$ such that $s' \leftrightarrow^*_{\mathcal{T}} s''\rho$ and $t' \leftrightarrow^*_{\mathcal{T}} t''\rho$ for some substitution ρ. As $\mathcal{S}$ is AC convergent for $\mathcal{T}$, s' and $s''\rho$ as well as t' and $t''\rho$ have the same $\mathcal{S}$-normal forms, which we denote by $\hat{s}$ and $\hat{t}$, respectively. We have $c_n(P') = \{(c(w,p,s'),\dots),(c(w',q,t'),\dots)\} \cup c_{\mathrm{AC}}(P')$ while the proof $Q\colon s' \leftrightarrow^*_{\mathcal{S}\cup\mathrm{AC}} s''\rho \leftrightarrow_{s''\approx t''} t''\rho \leftrightarrow^*_{\mathcal{S}\cup\mathrm{AC}} t'$ has cost $c_n(Q) = \{(\{\hat{s},\hat{t}\},\dots)\} \cup c_{\mathcal{S}\cup\mathrm{AC}}(Q)$, so $P' \Rightarrow_n Q$ holds because $w \succ s' \succ \hat{s}$ and $w' \succ t' \succ \hat{t}$. As $\mathrm{CP}_{\mathcal{L}}(\mathcal{R}_n, \mathcal{R}_n^e) \subseteq \bigcup_i \mathcal{E}_i$ the proof Q actually exists in some $(\mathcal{T}, \mathcal{E}_i, \mathcal{R}_i)$. By the Persistence Lemma 3.4 there is also a proof Q' in $\mathcal{T} \cup \mathcal{R}_n$ such that $P' \Rightarrow_n Q \Rightarrow_n^= Q'$.

Next, assume P contains a peak of the form (ii). If such a pattern originates from a non-overlap then by Lemma 3.6(c) it could be replaced by a smaller proof. Otherwise, by Lemma 3.5, the proof P must contain a proof corresponding to an AC-critical pair $s' \leftarrow \times \rightarrow t'$ in $\mathrm{CP}_{\mathrm{AC}}(\mathcal{R}_n, \mathcal{S}^e) \cup \mathrm{CP}_{\mathrm{AC}}(\mathcal{S}, \mathcal{R}_n)$. Then $s' \leftarrow \times \rightarrow t'$ must originate from an AC-critical peak Q of the form $s' \;{}_{r\leftarrow\ell}\!\leftarrow \cdot \leftrightarrow^*_{\mathrm{AC}} \cdot \rightarrow_{u\to v} t'$ between rules $\ell \to r \in \mathcal{R}_n$ and $u \to v \in \mathcal{S}$, and a proof Q' in $\mathcal{T} \cup \mathcal{R}_n$ satisfying $Q \Rightarrow_n Q'$ exists according to Lemma 3.12. This implies $P = P[Q] \Rightarrow_n P[Q']$.

If P contains a subproof Q of the form (iii) we have $c_n(Q) = \{(\{u,t\},\dots)\} \cup c_{\mathrm{AC}}(Q)$. Since $u \neq u{\downarrow}$ there is some step $u \rightarrow_{\mathcal{S}/\mathrm{AC}} s$, and thus a peak $P'\colon s \;{}_{\mathcal{S}/\mathrm{AC}}\!\leftarrow u \rightarrow_{\mathcal{R}_n/\mathrm{AC}} t$. If P' does not constitute a proper overlap then there exists a rewrite proof Q' of $s \approx t$ which contains only terms smaller than u. For $Q''\colon u \rightarrow_{\mathcal{S}/\mathrm{AC}} s$ the proof $Q''Q'$ is thus smaller than Q as $(\{u,t\},\dots) \in c_n(Q)$ dominates all cost tuples in $c_n(Q''Q')$. If P' constitutes a critical peak then by Lemma 3.12 there exists a proof Q' of $s \approx t$ such that $P' \Rightarrow_n Q'$ and for $(T,\dots) \in c_n(Q')$ all terms in T are smaller than u. Again $Q \Rightarrow_n Q''Q'$ holds.

Finally, if P contains a subproof of the form (iv) then AC convergence of $\mathcal{S}$ yields a smaller proof according to Lemma 3.6(a). ◀

▶ **Correctness Theorem 3.15.** *A fair and nonfailing* NKB *run succeeds.*

Proof. Let $(\mathcal{E}_0, \varnothing) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \cdots \vdash (\varnothing, \mathcal{R}_n)$ be the run under consideration. We show that $\leftrightarrow^*_{\mathcal{E}_0\cup\mathcal{T}} \subseteq \rightarrow^*_{\mathcal{R}\setminus\mathcal{S}} \cdot \leftrightarrow^*_{\mathcal{T}} \cdot \leftarrow^*_{\mathcal{R}\setminus\mathcal{S}}$. According to the Persistence Lemma 3.4, any pair of terms in $\leftrightarrow^*_{\mathcal{E}_0\cup\mathcal{T}}$ has a proof in $\mathcal{T} \cup \mathcal{R}_n$. Let P be such a proof which is minimal with respect to $\Rightarrow_n$, and assume it is not a normalized rewrite proof. By fairness there exists a proof Q in $(\mathcal{T}, \mathcal{E}_i, \mathcal{R}_i)$ for some $0 \le i \le n$ such that $P \Rightarrow_n Q$. According to persistence $Q \Rightarrow_n^= Q'$ for some Q' in $\mathcal{T} \cup \mathcal{R}_n$, contradicting the minimality of P. By the Soundness Lemma 3.3 the relations $\leftrightarrow^*_{\mathcal{T}\cup\mathcal{R}_n}$ and $\leftrightarrow^*_{\mathcal{T}\cup\mathcal{E}_0}$ coincide, so $\mathcal{R}_n$ is $\mathcal{S}$-convergent for $\mathcal{E}_0$. ◀

▶ **Example 3.16.** Consider an Abelian group with AC operator $\cdot$ and an endomorphism f as described by the following set of equations:

$$\mathsf{e}\cdot x \approx x \qquad \mathsf{i}(x)\cdot x \approx \mathsf{e} \qquad \mathsf{f}(x\cdot y) \approx \mathsf{f}(x)\cdot\mathsf{f}(y)$$

together with LPO with precedence $\mathsf{f} \succ \mathsf{i} \succ \cdot \succ \mathsf{e}$. We can obviously apply normalized completion with respect to AC, so $\mathcal{S} = \varnothing$. This results in the AC-convergent TRS $\mathcal{R}_{\mathrm{AC}}$:

$$\begin{array}{lll} \mathsf{e}\cdot x \to x & \mathsf{i}(x)\cdot x \to \mathsf{e} & \mathsf{i}(\mathsf{e}) \to \mathsf{e} \\ \mathsf{i}(\mathsf{i}(x)) \to x & \mathsf{i}(x\cdot y) \to \mathsf{i}(x)\cdot\mathsf{i}(y) & \mathsf{f}(x\cdot y) \to \mathsf{f}(x)\cdot\mathsf{f}(y) \\ \mathsf{f}(\mathsf{e}) \to \mathsf{e} & \mathsf{f}(\mathsf{i}(x)) \to \mathsf{i}(\mathsf{f}(x)) & \end{array}$$

Alternatively, we can consider $\mathcal{S}_{\mathrm{G}} = \{\mathsf{e} \cdot x \to x,\ \mathsf{i}(x) \cdot x \to \mathsf{e},\ \mathsf{i}(\mathsf{e}) \to \mathsf{e},\ \mathsf{i}(\mathsf{i}(x)) \to x,\ \mathsf{i}(x \cdot y) \to \mathsf{i}(x) \cdot \mathsf{i}(y)\}$ which is known to be an AC-convergent representation of Abelian groups [4]. Note that $\mathcal{S}_{\mathrm{G}} \subseteq \succ$. An NKB run with respect to $\mathcal{S}_{\mathrm{G}}$ results in the TRS $\mathcal{R}_{\mathrm{G}}$:

$$\mathsf{f}(x \cdot y) \to \mathsf{f}(x) \cdot \mathsf{f}(y) \qquad \mathsf{f}(\mathsf{e}) \to \mathsf{e} \qquad \mathsf{f}(\mathsf{i}(x)) \to \mathsf{i}(\mathsf{f}(x))$$

A TRS $\mathcal{R}$ is called $\mathcal{S}$*-reduced* if for all rules $\ell \to r$ in $\mathcal{R}$ the term r is in normal form with respect to $\to_{\mathcal{S}/\mathrm{AC}}$ and $\to_{\mathcal{R}\setminus\mathcal{S}}$, and ℓ is in normal form with respect to $\to_{\mathcal{S}/\mathrm{AC}}$ and $\to_{\ell' \to r' \setminus \mathcal{S}}$ for every rule $\ell' \to r'$ in $\mathcal{R}$ different from $\ell \to r$. A TRS $\mathcal{R}$ is $\mathcal{S}$*-canonical for* $\mathcal{E}$ if it is both $\mathcal{S}$-reduced and $\mathcal{S}$-convergent for $\mathcal{E}$, and a run is *simplifying* if simplify, compose and collapse are applied exhaustively. As two TRSs that are $\mathcal{S}$-canonical for $\mathcal{E}$ and contained in the same AC-compatible reduction order $\succ$ are equal up to variable renaming and AC equivalence [17], correctness implies the following completeness result.

▶ **Corollary 3.17.** *Assume $\mathcal{R}$ is a finite $\mathcal{S}$-canonical system for $\mathcal{E}$ and let $\succ$ be an AC-compatible reduction order that contains $\mathcal{R}$ and $\mathcal{S}$. Then any fair, nonfailing, and simplifying run from $\mathcal{E}$ using $\succ$ will produce an $\mathcal{S}$-canonical system $\mathcal{R}'$ such that $\mathcal{R}$ and $\mathcal{R}'$ are equal up to variable renaming and AC equivalence.*

For infinite runs one can show the stronger completeness result that whenever a finite $\mathcal{S}$-canonical system for some theory exists, any nonfailing run applying a corresponding reduction order succeeds in *finitely many steps* [26].

We conclude this section by commenting on the definition of normalizing pairs. In [19, Definition 3.5] and [20, Definition 3.1], normalizing pairs are defined as follows. Given terms u and v such that $u = u{\downarrow}$, $v = v{\downarrow}$, and $u \succ v$, the functions (Θ, Ψ) form an $\mathcal{S}$-normalizing pair if and only if

- **(i)** for any single-step proof $s \leftrightarrow_{u \approx v} t$ there is a proof P in $(\mathcal{T}, \Theta(u,v), \Psi(u,v))$ such that $s \leftrightarrow_{u \approx v} t \Rightarrow P$, and
- **(ii)** for all $\ell \to r \in \Psi(u,v)$, all sets of rules $\mathcal{R}$ and all r' such that $r \to^*_{\mathcal{R}\setminus\mathcal{S}} r'$ and any single-step irreducible[6] proof $s \to_{\ell \to r'} t$ there is a proof P in $(\mathcal{T}, \Theta(u,v), \Psi(u,v) \cup \mathcal{R})$ such that $s \to_{\ell \to r'} t \Rightarrow P$.

In our understanding four issues arise with this definition.

- **(a)** It does not require $\Theta(u,v)$ and $\Psi(u,v)$ to be part of the equational theory.
- **(b)** It does not guarantee termination of $\Psi(u,v)$ together with previously oriented rules.
- **(c)** Joinability of AC-critical pairs between $\mathcal{S}$ and $\Psi(u,v)$ is not ensured: Consider the simple example where the theory $\mathcal{E}_0 = \{x + \mathsf{a} \approx \mathsf{a}\}$ is to be completed with respect to $\mathcal{S} = \{y + \mathsf{b} \to \mathsf{b}\}$. We can choose $\Theta(x + \mathsf{a}, \mathsf{a}) = \varnothing$ and $\Psi(x + \mathsf{a}, \mathsf{a}) = \{x + \mathsf{a} \to \mathsf{a}\}$, satisfying (i) and (ii). We obtain the run $(\{x + \mathsf{a} \approx \mathsf{a}\}, \varnothing) \vdash (\varnothing, \{x + \mathsf{a} \to \mathsf{a}\})$ which is obviously fair. But $\{x + \mathsf{a} \to \mathsf{a}\}$ is not $\mathcal{S}$-convergent as the AC-critical pair $\mathsf{a} \leftarrow \rtimes \to \mathsf{b}$ between $\mathcal{S}$ and $x + \mathsf{a} \to \mathsf{a}$ is not considered.
- **(d)** The general normalizing pair [19, Definition 3.9] does not match this definition: Assume we orient $x + \mathsf{a} \approx \mathsf{a}$ as $x + \mathsf{a} \to \mathsf{a}$. The general normalizing pair sets $\Theta(x + \mathsf{a}, \mathsf{a}) = \mathrm{CP}_{\mathrm{AC}}(\mathcal{S}, x + \mathsf{a} \to \mathsf{a}) \cup \mathrm{CP}_{\mathrm{AC}}(x + \mathsf{a} \to \mathsf{a}, \mathcal{S}^e)$ and $\Psi(x + \mathsf{a}, \mathsf{a}) = \{x + \mathsf{a} \to \mathsf{a}\}$. Then property (ii) is not satisfied: for $\mathcal{R} = \varnothing$ and $r = r' = \mathsf{a}$ there exists no smaller proof than $x + \mathsf{a} \to^{\epsilon}_{x+\mathsf{a}\to\mathsf{a}} \mathsf{a}$ (and there is also no reason why such a proof should be necessary).

With the earlier definition in [17, Definition 4.4] similar issues arise, cf. [26]. Due to these ambiguities the notion of normalizing pairs was modified according to Definition 3.7.

[6] A proof is irreducible if it is minimal with respect to the proof reduction relation $\Rightarrow$.

3.2 Critical Pair Criteria

Critical pair criteria constitute a means to filter out critical pairs that can be ignored without compromising completeness. Let $\mathcal{L}$ be a theory between AC and $\mathcal{T}$. A critical pair criterion CPC maps $(\mathcal{E},\mathcal{R})$ to a set of equations such that $\mathrm{CPC}(\mathcal{E},\mathcal{R})$ is a subset of $\mathrm{CP}_{\mathcal{L}}(\mathcal{R},\mathcal{R}^e)$. As for standard completion, the *compositeness criterion* serves as a general condition.

▶ **Definition 3.18.** Let $\mathcal{E}$ be a set of equations and $\mathcal{R}$ be a set of rewrite rules. An equational proof P that has the form of a peak $s \leftarrow \cdot \leftrightarrow^*_{\mathcal{L}} \cdot \rightarrow t$ is *composite* in $(\mathcal{T},\mathcal{E},\mathcal{R})$ if there exist terms $u_0,\dots,u_{k+1}$ where $s = u_0$, $t = u_{k+1}$ and $u \succ u_j$ for all $0 \leqslant j \leqslant k+1$, and proofs $P_0,\dots,P_k$ in $(\mathcal{T},\mathcal{E},\mathcal{R})$ such that P_j proves $u_j \approx u_{j+1}$ and $P \succcurlyeq_n P_j$ for all $1 \leqslant j \leqslant k$, and any $n \geqslant 0$. The *compositeness criterion* $\mathrm{CCP}_{\mathcal{L}}(\mathcal{E},\mathcal{R})$ returns all $\mathcal{L}$-critical pairs among rules in $\mathcal{R}$ for which the associated overlaps are composite.

We now relax Lemma 3.14 by proving that composite critical pairs can safely be ignored.

▶ **Lemma 3.19.** *Consider a nonfailing* NKB *run* $\gamma\colon (\mathcal{E}_0,\varnothing) \vdash (\mathcal{E}_1,\mathcal{R}_1) \vdash \cdots \vdash (\varnothing,\mathcal{R}_n)$ *and let* $\mathcal{C}$ *be a subset of* $\bigcup_i \mathrm{CCP}(\mathcal{E}_i,\mathcal{R}_i)$. *If* $\mathrm{CP}_{\mathcal{L}}(\mathcal{R}_n,\mathcal{R}_n^e) \setminus \mathcal{C} \subseteq \bigcup_i \mathcal{E}_i$ *then* γ *is fair.*

Proof. Induction on $\succcurlyeq_n$ shows that any proof in $\mathcal{T} \cup \mathcal{R}_n$ can be transformed into a normalized rewrite proof. Any non-rewrite proof must contain (i) a peak $s \;{}_{\mathcal{R}_n/\mathrm{AC}}\!\leftarrow \cdot \rightarrow_{\mathcal{R}_n/\mathrm{AC}} t$, or (ii) a peak $s \;{}_{\mathcal{R}_n/\mathrm{AC}}\!\leftarrow \cdot \rightarrow_{\mathcal{S}/\mathrm{AC}} t$, or (iii) a subproof $u \rightarrow_{\mathcal{R}_n/\mathrm{AC}} t$ such that $u \neq u{\downarrow}$, or (iv) a peak $s \;{}_{\mathcal{S}/\mathrm{AC}}\!\leftarrow \cdot \rightarrow_{\mathcal{S}/\mathrm{AC}} t$. In the latter three cases existence of a smaller proof can be argued as in Lemma 3.14. This also holds for (i) if the peak is a non-overlap, or if it is a proper overlap and the respective critical pair occurs in $\bigcup_i \mathcal{E}_i$. In all these cases this smaller proof can thus be transformed into a rewrite proof by the induction hypothesis. It remains to consider the subcase of (i) where there are a proof $P\colon s \;{}_{\mathcal{R}_n/\mathrm{AC}}\!\leftarrow u \rightarrow_{\mathcal{R}_n/\mathrm{AC}} t$ and a critical pair $\ell \simeq r \in \mathrm{CP}_{\mathcal{L}}(\mathcal{R}_n,\mathcal{R}_n^e)$ such that $s \leftrightarrow^*_{\mathcal{L}} C[\ell\sigma] \leftrightarrow_{\ell\approx r} C[r\sigma] \leftrightarrow^*_{\mathcal{L}} t$ but $\ell \simeq r$ does not occur in any set $\mathcal{E}_i$. Hence we must have $\ell \simeq r \in \mathrm{CCP}(\mathcal{E}_i,\mathcal{R}_i)$ for some i. Let the corresponding critical overlap be $P'\colon \ell \leftarrow v \leftrightarrow^*_{\mathcal{L}} v' \rightarrow r$, so $P = P[C[P'\sigma]]$. By definition, there are terms $v_0,\dots,v_{k+1}$ such that $\ell = v_0$, $r = v_{k+1}$ and $v \succ v_j$ for all $0 \leqslant j \leqslant k+1$, and $(\mathcal{E}_i,\mathcal{R}_i)$ admits proofs P_j of $v_j \approx v_{j+1}$ which are smaller than P'. By the Persistence Lemma 3.4 there are respective proofs P'_j in $\mathcal{T} \cup \mathcal{R}_n$ such that $P_j \Rightarrow^{=}_n P'_j$. By the induction hypothesis all these proofs P'_j can be transformed into normalized rewrite proofs Q_j in $\mathcal{T} \cup \mathcal{R}_n$. Consequently all terms in the combined proof $Q\colon Q_0 \cdots Q_k$ of $\ell \approx r$ are smaller than v, so $P' \Rightarrow_n Q$ and hence $P = P[C[P'\sigma]] \Rightarrow_n P[C[Q\sigma]]$. Hence, as P can be transformed into a smaller proof it can be transformed into a normalized rewrite proof by the induction hypothesis. ◀

Although the compositeness criterion is very general, several special cases can be checked efficiently. Consider an overlap $\langle \ell_1 \rightarrow r_1, p, \ell_2 \rightarrow r_2 \rangle_\Sigma$ giving rise to the set of critical peaks

$$P\colon s \xleftarrow[r_1 \leftarrow \ell_1]{p,\sigma} u \overset{*}{\underset{\mathcal{L}}{\leftrightarrow}} u' \xrightarrow[\ell_2 \rightarrow r_2]{\epsilon,\sigma} t \tag{4}$$

such that $\sigma \in \Sigma$. If $u \neq u{\downarrow}$ or $u' \neq u'{\downarrow}$ then the $\mathcal{L}$-critical pair $s \leftarrow \bowtie \rightarrow t$ is *$\mathcal{S}$-reducible.*

Let us now assume that both u and u' in a peak (4) are in normal form with respect to $\rightarrow_{\mathcal{S}/\mathrm{AC}}$. By AC convergence of $\mathcal{S}$ and $\mathcal{L} \subseteq \mathcal{S}$ we thus have $u \leftrightarrow^*_{\mathrm{AC}} u'$. Now assume there is a rewrite step $u \leftrightarrow^*_{\mathrm{AC}} \cdot \rightarrow_{\mathcal{R}} v$ using a rule $\ell_3 \rightarrow r_3$ at position q, such that $(\ell_3 \rightarrow r_3, q)$ is different from $(\ell_1 \rightarrow r_1, p)$ and $(\ell_2 \rightarrow r_2, \epsilon)$. Thus there are proofs

$$P_1\colon s \xleftarrow[r_1 \leftarrow \ell_1]{p} u \overset{*}{\underset{\mathrm{AC}}{\leftrightarrow}} v' \xrightarrow[\ell_3 \rightarrow r_3]{q} v \qquad\qquad P_2\colon v \xleftarrow[r_3 \leftarrow \ell_3]{q} v' \overset{*}{\underset{\mathrm{AC}}{\leftrightarrow}} u' \xrightarrow[\ell_1 \rightarrow r_1]{\epsilon} t \tag{5}$$

such that P_1P_2 proves $s \approx t$. An AC-critical pair (4) is *not prime* if $u|_p \rhd_{\mathrm{AC}} v'|_q$.

▶ **Lemma 3.20.** *Any $\mathcal{L}$-critical pair which is $\mathcal{S}$-reducible or non-prime is composite.*

Proof. First we consider the case of an $\mathcal{L}$-critical pair. Let P be a peak of the form (4), and assume $u \to_{\mathcal{S}/\mathrm{AC}} v$. We thus also have another equational proof P_1P_2 of $s \approx t$, with

$$P_1\colon s \xleftarrow[r_1 \leftarrow \ell_1]{p,\sigma} u \xrightarrow[\mathcal{S}/\mathrm{AC}]{} v \qquad\qquad P_2\colon v \xleftarrow[\mathcal{S}/\mathrm{AC}]{} u \leftrightarrow^*_{\mathcal{L}} u' \xrightarrow[\ell_2 \to r_2]{\epsilon,\sigma} t$$

As u is $\mathcal{S}/\mathrm{AC}$-reducible we have $c(u', \epsilon, t) = \{u', t\}$, such that the proof costs amount to $c_n(P) = \{(c(u,p,s),\ldots), (\{u',t\},\ldots)\} \cup c_{\mathcal{L}}(P)$, $c_n(P_1) = \{(c(u,p,s),\ldots), (\bot,\ldots)\} \cup c_{\mathrm{AC}}(P_1)$, and $c_n(P_2) = \{(\bot,\ldots), (\{u',t\},\ldots)\} \cup c_{\mathrm{AC}}(P_2) \cup c_{\mathcal{L}}(P)$ where $c_{\mathcal{L}}(P)$ refers to the cost of the subproof $u \leftrightarrow^*_{\mathcal{L}} u'$ and $c_{\mathrm{AC}}(P_i)$ corresponds to the complexities of possibly required AC-steps in $u \to_{\mathcal{S}/\mathrm{AC}} v$. Note that the complexities of AC steps are smaller than the first two cost tuples in $c_n(P)$. We have $P \succcurlyeq_n P_1$ and $P \succcurlyeq_n P_2$, so the AC-critical pair is composite for NKB. A symmetric argument shows compositeness of any critical pair where u' is $\mathcal{S}/\mathrm{AC}$-reducible.

Let us now consider the case of a non-prime critical pair. As u, u', and v' are in $\mathcal{S}$-normal form, proof costs have the following shape, independent of n:

$$\begin{aligned} c_n(P) &= \{(\{u\}, \{u\}, u|_p, \ldots), (\{u'\}, \{u'\}, u', \ldots)\} \cup c_{\mathrm{AC}}(P) \\ c_n(P_1) &= \{(\{u\}, \{u\}, u|_p, \ldots), (\{v'\}, \{v'\}, v'|_q, \ldots)\} \cup c_{\mathrm{AC}}(P_1) \\ c_n(P_2) &= \{(\{u'\}, \{u'\}, u', \ldots), (\{v'\}, \{v'\}, v'|_q, \ldots)\} \cup c_{\mathrm{AC}}(P_2) \end{aligned}$$

From $u' \leftrightarrow^*_{\mathrm{AC}} v'$ we obtain $\{u'\} \succeq_{\mathsf{mul}} \{v'\}$. Therefore $u' \leftrightarrow^*_{\mathrm{AC}} u \trianglerighteq u|_p \rhd_{\mathrm{AC}} v'|_q$ and thus $u' \rhd_{\mathrm{AC}} v'|_q$, so we have $P \succcurlyeq_n P_1$ and $P \succcurlyeq_n P_1$. Furthermore, as $u|_p \rhd_{\mathrm{AC}} v'|_q$ we have $P \succcurlyeq_n P_2$ and $P \succcurlyeq_n P_2$ for any $n \geqslant 0$. It follows that P is composite. ◀

It can be shown that also the *connectedness criterion* proposed for standard completion [14] is applicable in normalized completion, and all these critical pair criteria are also compatible with a proof order based upon [19] and hence applicable in *infinite* runs, cf. [26].

4 Normalized Completion with Termination Tools

Classical Knuth-Bendix completion requires a fixed reduction order as input. To avoid fixing this critical parameter from the very beginning and obtain a greater variety of usable orders, Wehrman *et al.* [25] proposed *completion with termination tools*. In this section we take a similar approach to normalized completion.

The inference rules in Figure 2 describe normalized completion with termination tools (NKBtt). In the orient rule, (Θ, Ψ) is again assumed to form an $\mathcal{S}$-normalizing pair for the terms s and t. A sequence $(\mathcal{E}_0, \varnothing, \varnothing) \vdash (\mathcal{E}_1, \mathcal{R}_1, \mathcal{C}_1) \vdash (\mathcal{E}_2, \mathcal{R}_2, \mathcal{C}_2) \vdash \cdots$ of NKBtt inference steps is called a *run*. Before giving a correctness proof we illustrate NKBtt on an example.

▶ **Example 4.1.** Consider the initial set of equations $\mathcal{E}_0 = \{\mathsf{a} + x \approx \mathsf{b} + \mathsf{g}(\mathsf{a})\}$ where $+$ is an AC symbol with unit $\mathsf{0}$, such that the theory $\mathcal{T}$ can be represented by $\mathcal{S} = \{x + \mathsf{0} \to x\}$. Note that the given equation cannot be oriented with an AC-compatible simplification order. Thus any completion tool restricted to orders such as AC-RPO or AC-KBO [13] fails immediately. But termination tools can verify AC termination of the rule $\mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a})$ using e.g. AC dependency pairs [2]. Hence the equation $\mathsf{a} + x \approx \mathsf{b} + \mathsf{g}(\mathsf{a})$ can be oriented in an NKBtt run. When using ACU-normalizing pairs [19], this results in the state

$$\mathcal{E}_1\colon \quad \mathsf{a} + \mathsf{0} \approx \mathsf{b} + \mathsf{g}(\mathsf{a}) \qquad \mathcal{R}_1\colon \quad \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a}) \qquad \mathcal{C}_1\colon \quad \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a})$$

orient	$\dfrac{\mathcal{E} \uplus \{s \simeq t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \Theta(s,t), \mathcal{R} \cup \Psi(s,t), \mathcal{C}'}$	if $s = s{\downarrow}$, $t = t{\downarrow}$ and $\mathcal{C}' \cup \mathcal{S}$ is AC terminating for $\mathcal{C}' = \mathcal{C} \cup \Psi(s,t)$
deduce	$\dfrac{\mathcal{E}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}}$	if $s \approx t \in \mathrm{CP}_{\mathcal{L}}(\mathcal{R}, \mathcal{R}^e)$
delete	$\dfrac{\mathcal{E} \uplus \{s \approx t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E}, \mathcal{R}, \mathcal{C}}$	if $s \leftrightarrow^*_{\mathrm{AC}} t$
normalize	$\dfrac{\mathcal{E} \uplus \{s \approx t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{s{\downarrow} \approx t{\downarrow}\}, \mathcal{R}, \mathcal{C}}$	if $s \neq s{\downarrow}$ or $t \neq t{\downarrow}$
simplify	$\dfrac{\mathcal{E} \uplus \{s \simeq t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{s \simeq u\}, \mathcal{R}, \mathcal{C}}$	if $t \to_{\mathcal{R} \setminus \mathcal{S}} u$
compose	$\dfrac{\mathcal{E}, \mathcal{R} \uplus \{s \to t\}, \mathcal{C}}{\mathcal{E}, \mathcal{R} \cup \{s \to u\}, \mathcal{C}}$	if $t \to_{\mathcal{R} \setminus \mathcal{S}} u$
collapse	$\dfrac{\mathcal{E}, \mathcal{R} \uplus \{t \to s\}, \mathcal{C}}{\mathcal{E} \cup \{u \approx s\}, \mathcal{R}, \mathcal{C}}$	if $t \to_{\mathcal{R} \setminus \mathcal{S}} u$

Figure 2 $\mathcal{S}$-normalized completion with termination tools (NKBtt).

After normalizing $\mathsf{a} + 0$ to a, we have

$$\mathcal{E}_2\colon \quad \mathsf{a} \approx \mathsf{b} + \mathsf{g}(\mathsf{a}) \qquad \mathcal{R}_2\colon \quad \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a}) \qquad \mathcal{C}_2\colon \quad \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a})$$

Since $\mathcal{C}_2 \cup \{\mathsf{b} + \mathsf{g}(\mathsf{a}) \to \mathsf{a}\}$ is AC terminating, we may perform an orient step:

$$\mathcal{E}_3\colon \qquad \mathcal{R}_3\colon \quad \begin{matrix} \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a}) \\ \mathsf{b} + \mathsf{g}(\mathsf{a}) \to \mathsf{a} \end{matrix} \qquad \mathcal{C}_3\colon \quad \begin{matrix} \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a}) \\ \mathsf{b} + \mathsf{g}(\mathsf{a}) \to \mathsf{a} \end{matrix}$$

In a compose step, the new rule can be used to replace $\mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a})$ by $\mathsf{a} + x \to \mathsf{a}$. Three subsequent applications of deduce yield the state

$$\mathcal{E}_7\colon \begin{matrix} \mathsf{a} + \mathsf{g}(\mathsf{a}) \approx \mathsf{a} + \mathsf{a} \\ \mathsf{a} + \mathsf{a} \approx \mathsf{a} + \mathsf{b} \\ \mathsf{a} + \mathsf{a} \approx \mathsf{a} \end{matrix} \qquad \mathcal{R}_7\colon \quad \begin{matrix} \mathsf{a} + x \to \mathsf{a} \\ \mathsf{b} + \mathsf{g}(\mathsf{a}) \to \mathsf{a} \end{matrix} \qquad \mathcal{C}_7\colon \quad \begin{matrix} \mathsf{a} + x \to \mathsf{b} + \mathsf{g}(\mathsf{a}) \\ \mathsf{b} + \mathsf{g}(\mathsf{a}) \to \mathsf{a} \end{matrix}$$

All terms in $\mathcal{E}_7$ simplify to a, so the resulting trivial equations can be deleted. As all critical pairs among rules in $\mathcal{R}_7$ were already deduced the run is fair, so $\mathcal{R}_7$ is $\mathcal{S}$-convergent for $\mathcal{E}_0$.

Correctness of NKBtt relies on the following simulation lemma.

▶ **Lemma 4.2.**

1. *Any* NKBtt *run* $(\mathcal{E}_0, \varnothing, \varnothing) \vdash^n (\mathcal{E}_n, \mathcal{R}_n, \mathcal{C}_n)$ *admits an* NKB *run* $(\mathcal{E}_0, \varnothing) \vdash^n (\mathcal{E}_n, \mathcal{R}_n)$ *using the AC-compatible reduction order* $\to^+_{(\mathcal{C}_n \cup \mathcal{S})/\mathrm{AC}}$.
2. *If* $(\mathcal{E}_0, \varnothing) \vdash^n (\mathcal{E}_n, \mathcal{R}_n)$ *is a valid* NKB *run using an AC-compatible reduction order* $\succ$ *then there is also a valid* NKBtt *run* $(\mathcal{E}_0, \varnothing, \varnothing) \vdash^n (\mathcal{E}_n, \mathcal{R}_n, \mathcal{C}_n)$ *such that* $\mathcal{C}_n \subseteq \succ$. ◀

▶ **Correctness Theorem 4.3.** *Any finite nonfailing and fair* NKBtt *run succeeds.* ◀

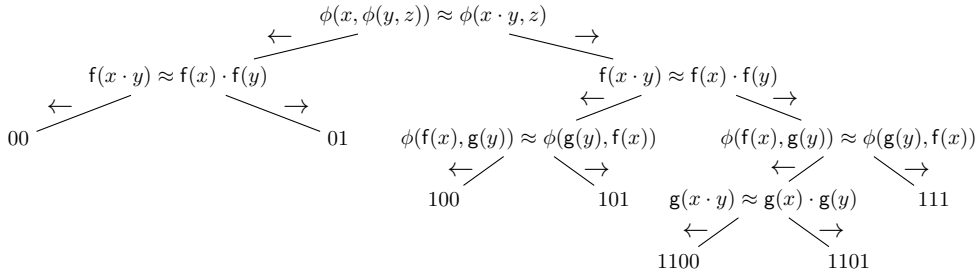


Figure 3 Part of the process tree developed in a run on CGA where process 1101 succeeds.

5 Implementation Details and Experimental Results

5.1 Multi-Completion

In completion with termination tools, the orient rule leaves a choice if the considered equation can be oriented in both directions. As the appropriate orientation of an equation is hard to predict, it is beneficial to keep track of multiple orientations. Thus, in our tool mkbtt we implemented a *multi-completion* variant of normalized completion with termination tools, following the approach suggested for completion with multiple reduction orders [15]. The basic idea is to simulate multiple NKBtt processes in parallel, but share common inferences to gain efficiency. Here a process corresponds to a sequence of decisions on how to orient equations. In our implementation, we model a process as a bit string. The initial process is denoted by ϵ. A formal description of this approach can be found in [26]. Here we content ourselves with giving an example.

▶ **Example 5.1.** We consider the system CGA describing an abelian group with a group action ϕ on itself such that two endomorphisms f and g commute with respect to ϕ:

$$\begin{array}{lll} x \cdot x^{-1} \approx \mathsf{e} & \mathsf{f}(x \cdot y) \approx \mathsf{f}(x) \cdot \mathsf{f}(y) & \mathsf{g}(x \cdot y) \approx \mathsf{g}(x) \cdot \mathsf{g}(y) \\ \phi(\mathsf{e}, x) \approx x & \phi(x, \phi(y, z)) \approx \phi(x \cdot y, z) & \phi(\mathsf{f}(x), \mathsf{g}(y)) \approx \phi(\mathsf{g}(y), \mathsf{f}(x)) \end{array}$$

together with the theory ACU, so $\mathcal{T} = \{x \cdot y \approx y \cdot x,\ (x \cdot y) \cdot z \approx x \cdot (y \cdot z),\ x \cdot \mathsf{e} \approx x\}$. Several equations are orientable in both directions. A multi-completion run thus gives rise to a process tree, where each branch corresponds to a possible sequence of orientations. Part of the process tree developed in a run on CGA run is shown in Figure 3. Note that the equation $\phi(\mathsf{f}(x), \mathsf{g}(y)) \approx \phi(\mathsf{g}(y), \mathsf{f}(x))$ cannot be oriented with AC-RPO or AC-compatible polynomial interpretations. Hence e.g. C*i*ME[7] cannot succeed, but by using muterm [1] for termination checks, mkbtt can produce an ACU-convergent system.

5.2 Implementation

We extended our tool mkbtt [28] to handle normalized multi-completion with termination tools. While the basic control loop remained the same, some changes had to be made to apply normalized completion. First of all, an AC-convergent TRS $\mathcal{S}$ representing the theory $\mathcal{T}$ is fixed and all terms are kept in $\mathcal{S}$-normalized form. The TRS $\mathcal{S}$ can be supplied by the user, otherwise mkbtt detects an applicable theory automatically (currently ACU, groups and

[7] We compared with C*i*ME 3.0.2, see `http://cime.lri.fr` and [7].

Table 1 Comparison of mkbtt using different theories.

	mkbtt								CiME[7]
	AC		ACU		AG		auto		
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)
abelian groups (AG)	1.6	77	2.4	61	0.1	5	0.1	5	0.05
AG + homomorphism	181.7	928	173.5	993	4.8	104	4.8	104	0.05
G0	1.9	82	1.9	70	0.1	8	0.1	8	?
G2	∞		∞		12.4	49	12.5	49	?
arithmetic	14.9	503	15.1	483	–		13.8	483	?
AC ring with unit	22.9	501	28.5	466	7.2	301	0.1	9	0.1
binary arithmetic	2.9	199	2.8	185	–		3.0	185	?
ternary arithmetic	18.1	816	17.3	781	–		17.3	781	?
Example 4.1	0.3	26	0.2	17	–		0.3	26	?
Example 5.1	∞		∞		15.4	486	15.2	486	?
Example 5.2	∞		∞		216.7	457	145.1	400	?
semiring	3.3	209	3.6	192	–		3.5	193	0.1
sum	1.4	4	1.5	5	–		1.4	4	?
completed systems	10		10		7		13		4

rings are supported, besides AC). We use general normalizing pairs, thus the orient inference had to be changed to add equations in the Θ component. Currently we always compute AC-critical pairs. In order to limit the number of nodes, the critical pair criteria described in Section 3.2 were implemented. Termination checks required in orient inference steps may be performed by an external termination tool supporting AC termination. Alternatively, mkbtt can also apply AC-RPO [23] or AC-KBO [13] internally. Further details can be found in [28] or obtained from the mkbtt website.[8]

5.3 Experiments

To evaluate our approach we ran mkbtt on problems collected from a number of different sources. All of the following tests were performed on an Intel Core Duo running at a clock rate of 1.4 GHz with 2.8 GB of main memory. Termination checks were done with muterm, and the primality critical pair criterion was used. The global timeout and the timeout for each termination check were set to 300 and 2 seconds, respectively.

In Table 1 we compare the results obtained with mkbtt applying different theories $\mathcal{T}$ (AC, AC with unit (ACU) and the theory of abelian groups (AG)) as well as automatic theory detection. The examples were collected from the literature, and some additional problems were added by the authors. The test set can be obtained from the website, where also the problems' sources are indicated. Columns (1) list the total time in seconds while columns (2) give the number of nodes created during the run. The symbol ∞ marks a timeout, and – indicates that the theory is not applicable. In line with [19], we observed that completion with respect to larger theories $\mathcal{T}$ is typically faster. Only in some cases such as the ring problem ACU-normalized completion is slower than AC-normalized completion, due to an unfortunate selection sequence. As expected, CiME is much faster if an appropriate reduction

[8] http://cl-informatik.uibk.ac.at/software/mkbtt

order is supplied as input. But as already mentioned, such a reduction order is hard to determine in advance, and in some cases no usable AC-RPO or polynomial interpretation exists. This is e.g. the case for Example 5.1, where mkbtt is able to find an ACU convergent system in a bit more than one hour, and for the example given below. When comparing AC-RPO with AC-KBO, there are some problems which can only be completed with the latter (e.g. binary arithmetic), but overall AC-RPO is more useful.

Concerning critical pair criteria, we found that the primality criterion decreased the total number of nodes by nearly 40%, which reduces the computation time by about 25%. $\mathcal{S}$-reducibility does not filter out any critical pairs if completion modulo ACU is performed. For normalized completion modulo group theory, very few redundant critical pairs are detected. The connectedness criterion was found to be comparatively expensive, and also the combined criterion could not achieve the same performance gain as the simpler primality criterion due to the additional effort of testing the criterion. Complete tables and more details on experimental results can be obtained from the website.

▶ **Example 5.2.** Consider ring theory with two commuting multiplicative mappings as defined by AC axioms for $+$ together with the equations

$$\begin{array}{rcc} x+0 \approx x & \mathsf{f}(1) \approx 1 & x \cdot (y+z) \approx (x \cdot y) + (x \cdot z) \\ x+(-x) \approx 0 & \mathsf{g}(1) \approx 1 & (x+y) \cdot z \approx (x \cdot z) + (x \cdot z) \\ 1 \cdot x \approx x & \mathsf{f}(x \cdot y) \approx \mathsf{f}(x) \cdot \mathsf{f}(y) & (x \cdot y) \cdot z \approx x \cdot (y \cdot z) \\ x \cdot 1 \approx x & \mathsf{g}(x \cdot y) \approx \mathsf{g}(x) \cdot \mathsf{g}(y) & \mathsf{f}(x) \cdot \mathsf{g}(y) \approx \mathsf{g}(y) \cdot \mathsf{f}(x) \end{array}$$

Our tool computes a convergent system using normalized completion modulo group theory/ring theory in 216.7/145.1 seconds producing 457/400 nodes, respectively. Normalized completion modulo AC and ACU yields a timeout. Due to the permutative equation $\mathsf{f}(x) \cdot \mathsf{g}(y) \approx \mathsf{g}(y) \cdot \mathsf{f}(x)$ no suitable input for CiME is known.

6 Conclusion

We considered finite normalized completion runs, and give correctness, completeness and uniqueness results using a slightly simpler proof order. Critical pair criteria for this setting were presented and proved correct using a relaxed notion of fairness. In order to tackle the limitation of a fixed reduction order, we proposed the use of automatic termination tools supporting AC-termination. Thus a user does not need to fix an AC-compatible reduction order in advance, a suitable ordering is instead found automatically. We implemented $\mathcal{S}$-normalized multi-completion with termination tools in mkbtt to evaluate our approach, which led to the construction of new convergent systems.

References

1 B. Alarcón, R. Gutiérrez, J. Iborra, and S. Lucas. Proving termination of context-sensitive rewriting with MU-TERM. In *6th PROLE*, volume 188 of *ENTCS*, pages 105–115, 2007.

2 B. Alarcón, S. Lucas, and J. Meseguer. A dependency pair framework for $A \vee C$-termination. In *8th WRLA*, volume 6381 of *LNCS*, pages 35–51, 2010.

3 F. Baader and T. Nipkow. *Term Rewriting and All That.* CUP, 1998.

4 L. Bachmair. *Canonical Equational Proofs.* Progress in Theoretical Computer Science. Birkhäuser, 1991.

5 L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. *Theoretical Computer Science*, 67(2,3):173–201, 1989.

6 L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *Journal of the ACM*, 41(2):236–276, 1994.

7 E. Contejean and C. Marché. C*i*ME: Complet*i*on modulo *E*. In *7th RTA*, volume 1103 of *LNCS*, pages 416–419, 1996.

8 G. Godoy and R. Nieuwenhuis. Paramodulation with built-in abelian groups. In *LICS 2000*, pages 413–424. IEEE Computer Society, 2000.

9 J.-P. Jouannaud. Confluent and coherent equational term rewriting systems: Application to proofs in abstract data types. In *8th CAAP*, volume 59 of *LNCS*, pages 269–283, 1983.

10 J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computation*, 15(4):1155–1194, 1986.

11 J.-P. Jouannaud and C. Marché. Termination and completion modulo associativity, commutativity and identity. *Theoretical Computer Science*, 104(1):29–51, 1992.

12 D.E. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

13 K. Korovin and A. Voronkov. An AC-compatible Knuth-Bendix order. In *19th CADE*, volume 2741 of *LNAI*, pages 47–59, 2003.

14 W. Küchlin. A confluence criterion based on the generalised Newman lemma. In *2nd EUROCAL*, volume 204 of *LNCS*, pages 390–399, 1985.

15 M. Kurihara and H. Kondo. Completion for multiple reduction orderings. *JAR*, 23(1):25–42, 1999.

16 D. Lankford and A.M. Ballantyne. Decision procedures for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions. Technical Report ATP-39, University of Texas, 1977.

17 C. Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité du problème du mot dans certaines classes de théories équationnelles*. PhD thesis, Université Paris-Sud, 1993.

18 C. Marché. Normalised rewriting and normalised completion. In *LICS 1994*, pages 394–403. IEEE Computer Society, 1994.

19 C. Marché. Normalized rewriting: An alternative to rewriting modulo a set of equations. *JSC*, 21(3):253–288, 1996.

20 C. Marché. Normalized rewriting: An unified view of Knuth-Bendix completion and Gröbner bases computation. *Progress in Computer Science and Applied Logic*, 15:193–208, 1998.

21 R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning*, pages 371–443. Elsevier Science Publishers, 2001.

22 G.E. Peterson and M.E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.

23 A. Rubio. A fully syntactic AC-RPO. *Information and Computation*, 178(2):515–533, 2002.

24 T. Sternagel, R. Thiemann, H. Zankl, and C. Sternagel. Recording completion for finding and certifying proofs in equational logic. In *Proc. 1st IWC*, pages 31–36, 2012.

25 I. Wehrman, A. Stump, and E.M. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *17th RTA*, volume 4098 of *LNCS*, pages 287–296, 2006.

26 S. Winkler. *Termination Tools in Automated Reasoning*. PhD thesis, University of Innsbruck, 2013.

27 S. Winkler and A. Middeldorp. Termination tools in ordered completion. In *5th IJCAR*, volume 6173 of *LNAI*, pages 518–532, 2010.

28 S. Winkler and A. Middeldorp. AC completion with termination tools (system description). In *23rd CADE*, volume 6803 of *LNAI*, pages 492–498, 2011.

29 S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara. Multi-completion with termination tools. *JAR*, 50(3):317–354, 2013.

Beyond Peano Arithmetic – Automatically Proving Termination of the Goodstein Sequence

Sarah Winkler*, **Harald Zankl**, and **Aart Middeldorp**

Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria
{sarah.winkler, harald.zankl, aart.middeldorp}@uibk.ac.at

Abstract

Kirby and Paris (1982) proved in a celebrated paper that a theorem of Goodstein (1944) cannot be established in Peano (1889) arithmetic. We present an encoding of Goodstein's theorem as a termination problem of a finite rewrite system. Using a novel implementation of ordinal interpretations, we are able to automatically prove termination of this system, resulting in the first automatic termination proof for a system whose derivational complexity is not multiple recursive. Our method can also cope with the encoding by Touzet (1998) of the battle of Hercules and Hydra, yet another system which has been out of reach for automated tools, until now.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases term rewriting, termination, automation, ordinals

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.335

1 Introduction

Since the beginning of the millennium there has been much progress regarding automated termination tools for rewrite systems.[1] Despite the many different techniques that have been developed, it seems that (terminating) TRSs which admit very long derivations are out of reach even for the most powerful tools. This is not surprising since many base methods induce rather small upper bounds on the derivational complexity. Hofbauer and Lautemann [14] have shown that polynomial interpretations are limited to double exponential derivational complexity. They further showed that the derivational complexity of a rewrite system compatible with KBO cannot be bounded by a primitive recursive function. Later, Lepper [19] established the Ackermann function as an upper bound for KBO, whereas Weiermann [30] proved a multiple recursive upper bound for LPO. More recently, Moser and Schnabl have studied upper bounds on the complexity when using these base methods in the dependency pair framework [25, 26]. Although dependency pairs significantly increase termination proving power, from the viewpoint of derivational complexity the limit is still multiple recursive. This has led to the conjecture [26, Conjecture 6.99] that for any system whose termination can be proved automatically by modern tools the length of its derivations can be bounded by a multiple recursive function (in the size of the starting terms).

In this paper we encode the computation of the sequences in Goodstein's theorem as a rewrite system $\mathcal{G}$ such that termination of $\mathcal{G}$ implies Goodstein's theorem. Since the latter is not provable in Peano arithmetic (Kirby and Paris [17]), the derivational complexity of $\mathcal{G}$ is not multiple recursive. Despite the fact that ordinals have been used in termination arguments since many decades [29, 11], until now a successful implementation for automatic

* Supported by a DOC-fFORTE fellowship of the Austrian Academy of Sciences.
[1] http://www.termination-portal.org/

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 335–351

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


termination proofs is lacking. In this paper we discuss automation of a termination criterion based on ordinal interpretations which is capable of proving $\mathcal{G}$ terminating, thereby disproving the above conjecture. Our implementation can also cope with Touzet's encoding [28] of the battle of Hercules and Hydra (also due to [17]), yet another system whose derivational complexity is not multiple recursive. In preliminary work [33, 31] we already used ordinal domains to increase automatic termination proving power. However, in [33] the focus in on string rewriting and the interpretation functions have a very limited shape to avoid ordinal arithmetic. As a consequence the method is limited to systems with at most multiple *exponential* derivational complexity. Similarly, [31] uses ordinal domains for generalized KBO, again for string rewriting only.

This paper is organized as follows. In the next section we recall ordinal arithmetic and weakly monotone algebras for termination proofs. In Section 3 we present our encoding of Goodstein's theorem and prove its correctness. Section 4 discusses how ordinal interpretations can be automated. Rewrite systems encoding the Hydra battle are the topic of Section 5, in which also the limitations of our implementation of ordinal interpretations become apparent. We conclude in Section 6.

2 Preliminaries

We recall some preliminaries about ordinal numbers. Ordinals are transitive sets well-ordered with respect to $\in$. Hence $\alpha < \beta$ if and only if $\alpha \in \beta$. By identifying $\varnothing$, $\{\varnothing\}$, $\{\varnothing, \{\varnothing\}\}$, ... with 0, 1, 2, ..., the natural numbers are embedded in the ordinals. If α is an ordinal then the ordinal $\alpha \cup \{\alpha\}$ is its successor, denoted by $\alpha + 1$. An ordinal β constitutes a successor ordinal if there is some α such that $\beta = \alpha + 1$, otherwise β is called a limit ordinal. For instance 1, 2, 3, ... are successor ordinals, whereas 0 and the smallest infinite ordinal ω are limit ordinals. The latter is equivalent to the set of all natural numbers. The following ordinal arithmetic operations constitute extensions of the respective operations on natural numbers (see [16] for details).

▶ **Definition 1.** For ordinals α and β their *sum* $\alpha + \beta$ is defined by recursion over β as (a) $\alpha + 0 = \alpha$, (b) $\alpha + \beta = (\alpha + \gamma) + 1$ if $\beta = \gamma + 1$, and (c) $\alpha + \beta = \bigcup_{\gamma<\beta} \alpha + \gamma$ if β is a positive limit ordinal.

Addition satisfies associativity $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$ but is not commutative, e.g., $1 + \omega = \omega \neq \omega + 1$.

▶ **Definition 2.** For ordinals α and β their *product* $\alpha \cdot \beta$ is defined by recursion over β as (a) $\alpha \cdot 0 = 0$, (b) $\alpha \cdot \beta = \alpha \cdot \gamma + \alpha$ if $\beta = \gamma + 1$, and (c) $\alpha \cdot \beta = \bigcup_{\gamma<\beta} \alpha \cdot \gamma$ if β is a positive limit ordinal.

Since $2 \cdot \omega = \omega \neq \omega \cdot 2$ multiplication is not commutative, and as $(\omega + 1) \cdot 2 = (\omega + 1) + (\omega + 1) = \omega \cdot 2 + 1$ also not right-distributive, but associativity $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$ and left-distributivity $\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma)$ hold. We mostly write αa for $\alpha \cdot a$.

▶ **Definition 3.** For ordinals α and β, recursion over β allows to define *exponentiation* α^β as follows: (a) $\alpha^0 = 1$, (b) $\alpha^\beta = \alpha^\gamma \cdot \alpha$ if $\beta = \gamma + 1$, and (c) $\alpha^\beta = \bigcup_{\gamma<\beta} \alpha^\gamma$ if β is a positive limit ordinal.

Examples of infinite ordinals include $\omega^1 = \omega$, $\omega\,3 = \omega + \omega + \omega$, $\omega^2 = \omega \cdot \omega$, $\omega^{\omega+1}$, and ω^{ω^ω}. The ordinal ϵ_0 is the smallest ordinal α which satisfies $\alpha^\omega = \alpha$. Let $\mathbb{O}$ denote the class of ordinal numbers smaller than ϵ_0, $\mathbb{N}$ ordinal numbers smaller than ω (the natural numbers), $>$ the standard order on ordinals, and $\geqslant$ its reflexive closure.

Recall that every ordinal $\alpha < \epsilon_0$ can be represented in Cantor normal form (CNF), i.e.,

$$\alpha = \omega^{\alpha_1} a_1 + \cdots + \omega^{\alpha_n} a_n \tag{1}$$

such that $\alpha_1 > \cdots > \alpha_n$ are in CNF as well and $a_1, \ldots, a_n \in \mathbb{N}_{>0}$. The ordinal 0 is represented as the empty sum.

▶ **Definition 4.** Let $\alpha = \omega^{\alpha_1} a_1 + \cdots + \omega^{\alpha_n} a_n$ and $\beta = \omega^{\beta_1} b_1 + \cdots + \omega^{\beta_m} b_m$ be ordinals in CNF, and $\{\gamma_1, \ldots, \gamma_k\} = \{\alpha_1, \ldots, \alpha_n\} \cup \{\beta_1, \ldots, \beta_m\}$ such that $\gamma_1 > \cdots > \gamma_k$. The *natural sum* of α and β is defined as $\alpha \oplus \beta = \omega^{\gamma_1}(a'_1 + b'_1) + \cdots + \omega^{\gamma_k}(a'_k + b'_k)$ where $a'_i = a_j$ ($b'_i = b_j$) if $\gamma_i = \alpha_j$ ($\gamma_i = \beta_j$) for some j, and $a'_i = 0$ ($b'_i = 0$) otherwise.

In contrast to standard addition, natural addition on ordinals enjoys all properties known from addition on natural numbers, e.g., $2 \oplus \omega = \omega \oplus 2 = \omega + 2$. For ordinal interpretations as considered later in this paper it is crucial that addition, natural addition, multiplication, and exponentiation are weakly monotone in both arguments.

We assume familiarity with term rewriting and termination in particular [27]. By $\overline{x}$ we abbreviate $x_1, \ldots, x_n$. We consider well-founded algebras $\mathcal{A}$ where the interpretation functions $f_{\mathcal{A}}$ take the following very general shape. *Ordinal interpretations* over variables $\overline{x}$ are the smallest set of expressions containing $\mathbb{N}$ and x_i for all $1 \leqslant i \leqslant n$, they are closed under (standard and natural) addition and multiplication, composition, and $\omega^{(\cdot)}$. An interpretation function $f_{\mathcal{A}}$ is *weakly monotone* if $a > b$ implies $f_{\mathcal{A}}(\ldots, a_{i-1}, a, a_{i+1}, \ldots) \geqslant f_{\mathcal{A}}(\ldots, a_{i-1}, b, a_{i+1}, \ldots)$. It is *simple* if $f_{\mathcal{A}}(a_1, \ldots, a_n) \geqslant a_i$ for all $1 \leqslant i \leqslant n$. An algebra is *simple/weakly monotone* if all its interpretation functions are. A TRS $\mathcal{R}$ is *compatible* with an algebra $\mathcal{A}$ if $[\alpha]_{\mathcal{A}}(\ell) > [\alpha]_{\mathcal{A}}(r)$ for every $\ell \to r \in \mathcal{R}$ and assignment α (also written $\mathcal{R} \subseteq >_{\mathcal{A}}$). Algebras may yield termination proofs.

▶ **Theorem 5** ([28, 34]). *A TRS is terminating if it is compatible with a well-founded weakly monotone simple algebra.* ◀

In order to prove termination of TRSs with non-multiple recursive derivation length, ordinal interpretations can be lexicographically combined with linear polynomial interpretations and matrix interpretations [9].

3 The Goodstein Sequence

In this section we present a TRS for the Goodstein sequence. Given $n > 1$, a natural number α is in *hereditary base n representation*, which we indicate by writing $(\alpha)_n$, if

$$(\alpha)_n = n^{(\alpha_k)_n} \cdot a_k + n^{(\alpha_{k-1})_n} \cdot a_{k-1} + \cdots + n^{(\alpha_0)_n} \cdot a_0 \tag{2}$$

such that $(\alpha_k)_n > \cdots > (\alpha_0)_n$ are in hereditary base n representation and $0 < a_i < n$ for all $0 \leqslant i \leqslant k$. For $m > n$ we denote by $(\alpha)_n^m$ the result of replacing n by m in $(\alpha)_n$, so $(\alpha)_n^m = m^{(\alpha_k)_n^m} \cdot a_k + m^{(\alpha_{k-1})_n^m} \cdot a_{k-1} + \cdots + m^{(\alpha_1)_n^m} \cdot a_1 + a_0$ is in hereditary base m representation.

▶ **Definition 6.** The *Goodstein sequence* g_α with starting value α is defined by $g_\alpha(0) = \alpha$ and $g_\alpha(i+1) = (g_\alpha(i))_{i+2}^{i+3} - 1$ for all $i \geqslant 0$.

▶ **Theorem 7** (Goodstein [12]). *For all α there exists a k such that $g_\alpha(k) = 0$.* ◀

By $G(\alpha)$ we denote the smallest number k with this property. Totality of this function is not provable in Peano arithmetic, as shown by Kirby and Paris [17]. Cichon [5] presented a very short proof using results concerning recursion theoretic hierarchies of functions. In particular, he showed that the growth rate of G is strongly related to H_{ϵ_0}.[2]

▶ **Definition 8.** For all $n > 1$ we define a mapping $[\cdot]_n$ to represent natural numbers in base n as ground terms over $\{\mathsf{c}, \mathsf{0}\}$, where c is a binary function symbol and $\mathsf{0}$ a constant. Let $(\alpha)_n$ be a natural number in hereditary base n representation as in (2). We denote the term $\mathsf{c}(x, \mathsf{c}(x, \cdots \mathsf{c}(x, y) \cdots))$ containing $k \geqslant 0$ occurrences of c by $\mathsf{c}^k(x, y)$. In particular, $\mathsf{c}^0(x, y) = y$. Then $[\cdot]_n$ is recursively defined such that $[0]_n = \mathsf{0}$ and

$$[\alpha]_n = \mathsf{c}^{a_0}([\alpha_0]_n, \dots \mathsf{c}^{a_{k-1}}([\alpha_{k-1}]_n, \mathsf{c}^{a_k}([\alpha_k]_n, \mathsf{0})) \dots)$$

Intuitively, given base n, the term $\mathsf{c}([\alpha]_n, [\beta]_n)$ represents the number $n^\alpha + \beta$, and terms contributing to the base n representation of a number are combined in increasing order.

▶ **Example 9.** For $(1)_2 = 2^0$ we have $[1]_2 = \mathsf{c}(\mathsf{0}, \mathsf{0})$, for $(2)_2 = 2^{2^0}$ we have $[2]_2 = \mathsf{c}(\mathsf{c}(\mathsf{0}, \mathsf{0}), \mathsf{0})$, for $(7)_2 = 2^{2^{2^0}} + 2^{2^0} + 2^0$ we have $[7]_2 = \mathsf{c}(\mathsf{0}, \mathsf{c}(\mathsf{c}(\mathsf{0}, \mathsf{0}), \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{0}, \mathsf{0}), \mathsf{0}), \mathsf{0}))))$, and for $(7)_3 = 3^{3^0} \cdot 2 + 3^0$ we have $[7]_3 = \mathsf{c}(\mathsf{0}, \mathsf{c}(\mathsf{c}(\mathsf{0}, \mathsf{0}), \mathsf{c}(\mathsf{c}(\mathsf{0}, \mathsf{0}), \mathsf{0})))$.

The following definition is inspired by Touzet's encoding of the Hydra battle [28] (see Example 22).

▶ **Definition 10.** Consider the following TRS $\mathcal{G}$ over a signature consisting of unary function symbols $\bullet$, $[]$, $\circ$ and binary function symbols f, h, in addition to $\mathsf{0}$ and c:

$$[] \circ x \to \circ [] x \tag{A1}$$
$$\bullet [] x \to [] \bullet \bullet x \tag{A2}$$
$$\circ x \to \bullet [] x \tag{A3}$$
$$\mathsf{c}(\mathsf{0}, x) \to \circ x \tag{B1}$$
$$\bullet \mathsf{c}(\mathsf{c}(x, y), z) \to \bullet \mathsf{f}(\mathsf{c}(x, y), z) \tag{B2}$$
$$\bullet \mathsf{f}(\mathsf{0}, x) \to \circ x \tag{C1}$$
$$\bullet \mathsf{f}(\mathsf{c}(x, y), z) \to \mathsf{h}(\bullet \mathsf{f}(x, y), \bullet \bullet \mathsf{f}(\mathsf{f}(x, y), z)) \tag{C2}$$
$$\bullet \mathsf{h}(x, y) \to \mathsf{h}(\bullet x, \bullet \bullet \mathsf{c}(x, y)) \tag{D1}$$
$$\mathsf{h}(x, y) \to \circ y \tag{D2}$$
$$\bullet \mathsf{f}(x, y) \to \mathsf{f}(\bullet x, y) \tag{E1}$$
$$\bullet \mathsf{c}(x, y) \to \mathsf{c}(\bullet x, \bullet y) \tag{E2}$$
$$\bullet x \to x \tag{E3}$$
$$\circ x \to x \tag{E4}$$

The idea is to encode the current base n as $[]^n$, followed by a term $[\alpha]_n$. The marker symbols $\circ$ and $\bullet$ are used to trigger rewrite steps while f and h compute intermediate results.

According to the following theorem, $\mathcal{G}$ simulates for any starting value the computation of the Goodstein sequence. Since the term $\bullet []^n [0]_n = \bullet []^n \mathsf{0}$ is clearly terminating, it follows that Theorem 7 is a consequence of the termination of $\mathcal{G}$.

[2] Here H is the Hardy function: $H_0(n) = n + 1$, $H_{\alpha+1}(n) = H_\alpha(n + 1)$, and $H_\lambda(n) = H_{\lambda_n}(n)$.

▶ **Theorem 11.** *Let $\alpha, n \in \mathbb{N}$ such that $\alpha > 0$ and $n > 1$. Then $\bullet\,[]^n\,[\alpha]_n \to_{\mathcal{G}}^+ \bullet\,[]^{n+1}\,[\beta]_{n+1}$ where $\beta = (\alpha)_n^{n+1} - 1$.*

The proof of this result requires some auxiliary facts about $\mathcal{G}$.

▶ **Lemma 12.**

(a) $\bullet^n\,\mathsf{h}(s,t) \to_{\mathcal{G}}^+ \circ\,\mathsf{c}^n(\bullet^n\,s, \bullet^{2n}\,t)$ *for all terms s and t.*

(b) *Let $\alpha, \beta \in \mathbb{N}$ and $n \in \mathbb{N}$ such that $n > 1$, $\beta + n^\alpha$ is positive, $s = [\alpha]_n$ and $t = [\beta]_n$. Then $\bullet^n\,\mathsf{f}(s,t) \to_{\mathcal{G}}^+ \circ\,u$ where $u = [\beta + n^\alpha - 1]_n$.*

Proof.

(a) By induction on n. If $n = 0$ then $\mathsf{h}(s,t) \to_{\mathcal{G}} \circ\,t$ in a single step using (D2). If $n > 0$ then

$$\begin{aligned}
\bullet^{n+1}\,\mathsf{h}(s,t) &\to_{\mathcal{G}} \bullet^n\,\mathsf{h}(\bullet\,s, \bullet\bullet\,\mathsf{c}(s,t)) && \text{(D1)}\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^n(\bullet^{n+1}\,s, \bullet^{2(n+1)}\,\mathsf{c}(s,t)) && (\star)\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^n(\bullet^{n+1}\,s, \mathsf{c}(\bullet^{2(n+1)}\,s, \bullet^{2(n+1)}\,t)) && \text{(E2)}\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^n(\bullet^{n+1}\,s, \mathsf{c}(\bullet^{n+1}\,s, \bullet^{2(n+1)}\,t)) && \text{(E3)}\\
&= \circ\,\mathsf{c}^{n+1}(\bullet^{n+1}\,s, \bullet^{2(n+1)}\,t)
\end{aligned}$$

where $(\star)$ applies the induction hypothesis.

(b) By induction on α. If $\alpha = 0$ then $[\alpha]_n = \mathsf{0}$ and $\bullet^n\,\mathsf{f}(\mathsf{0},t) \to_{\mathcal{G}} \bullet^{n-1} \circ t \to_{\mathcal{G}}^* \circ\,t$ using rules (C1) and (E3). Since $\beta + n^0 - 1 = \beta$ and $t = [\beta]_n$ the claim holds. If $\alpha > 0$ then $[\alpha]_n = \mathsf{c}(s',t')$ and $s' = [\gamma]_n$ and $t' = [\delta]_n$ for some $\gamma, \delta \in \mathbb{N}$, so $\alpha = \delta + n^\gamma$. We have

$$\begin{aligned}
\bullet^n\,\mathsf{f}(\mathsf{c}(s',t'),t) &\to_{\mathcal{G}} \bullet^{n-1}\,\mathsf{h}(\bullet\mathsf{f}(s',t'), \bullet\bullet\,\mathsf{f}(\mathsf{f}(s',t'),t)) && \text{(C2)}\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^{n-1}(\bullet^n\,\mathsf{f}(s',t'), \bullet^{2n}\,\mathsf{f}(\mathsf{f}(s',t'),t)) && \text{(a)}\\
&\to_{\mathcal{G}}^* \circ\,\mathsf{c}^{n-1}(\bullet^n\,\mathsf{f}(s',t'), \bullet^n\,\mathsf{f}(\bullet^n\,\mathsf{f}(s',t'),t)) && \text{(E1)}\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^{n-1}(\circ\,w, \bullet^n\,\mathsf{f}(\circ\,w,t)) && (\star)\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^{n-1}(w, \bullet^n\,\mathsf{f}(w,t)) && \text{(E4)}\\
&\to_{\mathcal{G}}^+ \circ\,\mathsf{c}^{n-1}(w, \circ\,w') && (\star\star)\\
&\to_{\mathcal{G}} \circ\,\mathsf{c}^{n-1}(w,w') && \text{(E4)}
\end{aligned}$$

where in $(\star)$ we apply the induction hypothesis since $\gamma < \alpha$ and so we obtain a term $w = [\delta + n^\gamma - 1]_n$. Since $\delta + n^\gamma - 1 < \alpha$, we can apply the induction hypothesis again in step $(\star\star)$, which yields a term w' such that $w' = \big[\beta + n^{\delta + n^\gamma - 1} - 1\big]_n$. Let $\nu = \delta + n^\gamma - 1$. For the term $v = \mathsf{c}^{n-1}(w,w')$ we thus have

$$v = [\beta + n^\nu \cdot (n-1) + n^\nu - 1]_n = \big[\beta + n^{\nu+1} - 1\big]_n = [\beta + n^\alpha - 1]_n \qquad \blacktriangleleft$$

Proof of Theorem 11. Since $\alpha > 0$, we have $[\alpha]_n = \mathsf{c}(s,t)$ for some terms s and t. We apply case analysis on s. If $s = \mathsf{0}$ then $t = [\alpha - 1]_n$ and we have

$$\begin{aligned}
\bullet\,[]^n\,\mathsf{c}(\mathsf{0},t) &\to_{\mathcal{G}} []^n\,\mathsf{c}(\mathsf{0},t) && \text{(E3)}\\
&\to_{\mathcal{G}} []^n \circ t && \text{(B1)}\\
&\to_{\mathcal{G}}^+ \circ\,[]^n\,t && \text{(A1)}\\
&\to_{\mathcal{G}} \bullet\,[]^{n+1}\,t && \text{(A3)}
\end{aligned}$$

Otherwise, $s = \mathsf{c}(u,v)$ so let $\mathsf{c}(u,v) = [\gamma]_n$ and $t = [\delta]_n$ for some $\gamma, \delta \in \mathbb{N}$. There is the following rewrite sequence:

$$\begin{aligned}
\bullet\, []^n\, \mathsf{c}(\mathsf{c}(u,v),t) &\to^+_{\mathcal{G}} []^n\, \bullet^{2^n}\, \mathsf{c}(\mathsf{c}(u,v),t) && \text{(A2)}\\
&\to^*_{\mathcal{G}} []^n\, \bullet^{n+1}\, \mathsf{c}(\mathsf{c}(u,v),t) && \text{(E3)}\\
&\to^*_{\mathcal{G}} []^n\, \bullet^{n+1}\, \mathsf{f}(\mathsf{c}(u,v),t) && \text{(B2)}\\
&\to^+_{\mathcal{G}} []^n \circ w && (\star)\\
&\to^+_{\mathcal{G}} \circ []^n\, w && \text{(A1)}\\
&\to_{\mathcal{G}} \bullet\, []^{n+1}\, w && \text{(A3)}
\end{aligned}$$

where $(\star)$ applies Lemma 12(b), according to which $w = [\delta + (n+1)^\gamma - 1]_{n+1}$. ◀

▶ **Theorem 13.** *The TRS $\mathcal{G}$ is terminating.*

Proof. We show termination of $\mathcal{G}$ according to Theorem 5. Consider the following interpretation $\mathcal{A}$ over the well-founded domain $\mathbb{O} \times \mathbb{N} \times \mathbb{N}$:

$$\begin{aligned}
0_{\mathcal{A}} &= (0,0,0) & []_{\mathcal{A}}(x,m,n) &= (x, 2m+2, n)\\
\mathsf{c}_{\mathcal{A}}((x,m,n),(y,k,l)) &= (\omega^x \oplus y + 1, 0, 0) & \circ_{\mathcal{A}}(x,m,n) &= (x, 2m+3, n)\\
\mathsf{f}_{\mathcal{A}}((x,m,n),(y,k,l)) &= (\omega^x \oplus y, 0, 0) & \bullet_{\mathcal{A}}(x,m,n) &= (x, m, n+m+1)\\
\mathsf{h}_{\mathcal{A}}((x,m,n),(y,k,l)) &= (y + \omega^{x+1}, 0, 0)
\end{aligned}$$

This interpretation is simple and weakly monotone. Because

$$\begin{aligned}
(x, 4m+8, n) &> (x, 4m+7, n) && \text{(A1)}\\
(x, 2m+2, 2m+n+3) &> (x, 2m+2, 2m+n+2) && \text{(A2)}\\
(x, 2m+3, n) &> (x, 2m+2, n+2m+3) && \text{(A3)}\\
(x+2, 0, 0) &> (x, 2m+3, n) && \text{(B1)}\\
(\omega^{\omega^x \oplus y+1} \oplus z + 1, 0, 1) &> (\omega^{\omega^x \oplus y+1} \oplus z, 0, 1) && \text{(B2)}\\
(x+1, 0, 1) &> (x, 2m+3, n) && \text{(C1)}\\
(\omega^{\omega^x \oplus y+1} \oplus z, 0, 1) &> (z + \omega^{\omega^x \oplus y+1}, 0, 0) && \text{(C2)}\\
(y + \omega^{x+1}, 0, 1) &> (y + \omega^{x+1}, 0, 0) && \text{(D1)}\\
(y + \omega^{x+1}, 0, 0) &> (y, 2k+3, l) && \text{(D2)}\\
(\omega^x \oplus y, 0, 1) &> (\omega^x \oplus y, 0, 0) && \text{(E1)}\\
(\omega^x \oplus y + 1, 0, 1) &> (\omega^x \oplus y + 1, 0, 0) && \text{(E2)}\\
(x, m, n+m+1) &> (x, m, n) && \text{(E3)}\\
(x, 2m+3, n) &> (x, m, n) && \text{(E4)}
\end{aligned}$$

it strictly orients all rules of $\mathcal{G}$. Hence $\mathcal{G}$ is terminating. ◀

4 Automation

In order to automate the search for suitable ordinal interpretations, we restrict to interpretations of a certain shape (see Definition 14). In Section 4.1 we show how for a given (parametric) algebra of this shape one can derive over- and underapproximations for a

term's interpretation, and encode the constraints on the (coefficients of the) interpretation as a problem in non-linear integer arithmetic for which suitable SMT solvers exist (see [32]). In contrast to other termination criteria, ordinal arithmetic (non-commutative, expressions may be consumed) significantly complicates the encoding. Section 4.2 elaborates on implementation issues needed for a successful automation.

In the sequel we consider ordinal expressions of the following shape.

▶ **Definition 14.** A *restricted ordinal expression* (*ROE*) over variables $\overline{x}$ is either 0 or[3]

$$\sum_{1\leqslant i\leqslant n} x_i f_i + \omega^{f'(\overline{x})} f_\omega \oplus \bigoplus_{1\leqslant i\leqslant n} x_i \hat{f}_i \oplus f_0 \tag{3}$$

where $f_0, f_1, \ldots, f_n, \hat{f}_1, \ldots, \hat{f}_n, f_\omega$ are (unknowns over the) naturals and $f'(\overline{x})$ is an ROE over $\overline{x}$. The *depth* of an ROE is the height of the tower of ω's. An *ROE algebra* is an algebra $\mathcal{O}$ in which for every n-ary function symbol f the interpretation function $f_\mathcal{O}$ is an ROE over $\overline{x}$.

4.1 Encodings

Let $f(\overline{x})$ and $g(\overline{x})$ be ROEs of the form

$$f(\overline{x}) = \sum_{1\leqslant i\leqslant n} x_i f_i + \omega^{f'(\overline{x})} f_\omega \oplus \bigoplus_{1\leqslant i\leqslant n} x_i \hat{f}_i \oplus f_0 \tag{4}$$

$$g(\overline{x}) = \sum_{1\leqslant i\leqslant n} x_i g_i + \omega^{g'(\overline{x})} g_\omega \oplus \bigoplus_{1\leqslant i\leqslant n} x_i \hat{g}_i \oplus g_0 \tag{5}$$

We assume that these expressions depend on the same variables $\overline{x}$ (otherwise the respective coefficients can be set to 0), and that variables appear in the same order (Section 4.2.2 explains how this is ensured). We first encode some auxiliary properties of (parametric) interpretations.

Let $\mathrm{zero}(f(\overline{x}))$ be true if and only if $f(\overline{x}) = 0$ or all of f_0, f_i, $\hat{f}_i$ and f_ω are 0. Let $c_i = \max(f_i, g_i)$ for all $i \in \{0, \ldots, n, \omega\}$. An upper bound $\mathrm{omax}(f, g)(\overline{x})$ is then given by $\mathrm{omax}(f, 0)(\overline{x}) = \mathrm{omax}(0, f)(\overline{x}) = f(\overline{x})$ and

$$\mathrm{omax}(f, g)(\overline{x}) = \sum_{1\leqslant i\leqslant n} x_i c_i + \omega^{\mathrm{omax}(f',g')(\overline{x})} c_\omega \oplus \bigoplus_{1\leqslant i\leqslant n} x_i \max(\hat{f}_i, \hat{g}_i) \oplus c_0$$

otherwise. For instance, if $f(\overline{x}) = x_1 + \omega^{x_2+1} \oplus x_3$ and $g(\overline{x}) = \omega^{x_1} 2 \oplus x_2 + 1$ then $\mathrm{omax}(f, g)(\overline{x}) = x_1 + \omega^{x_1+x_2+1} 2 \oplus x_2 \oplus x_3 + 1$. Clearly, $[\alpha](f(\overline{x})) \leqslant [\alpha](\mathrm{omax}(f, g)(\overline{x}))$ and $[\alpha](g(\overline{x})) \leqslant [\alpha](\mathrm{omax}(f, g)(\overline{x}))$ for all assignments α. Consideration of a variable x_i by $f(\overline{x})$ can be recursively encoded as follows:

$$\mathrm{con}(x_i, f(\overline{x})) = \begin{cases} \bot & \text{if } f(\overline{x}) = 0 \\ f_i > 0 \vee \hat{f}_i > 0 \vee (\mathrm{con}(x_i, f'(\overline{x})) \wedge f_\omega > 0) & \text{otherwise} \end{cases}$$

If $f(\overline{x})$ and $g(\overline{x})$ are defined as above then $\mathrm{con}(x_i, f(\overline{x})) = \top$ for all $1 \leqslant i \leqslant 3$ and $\mathrm{con}(x_j, g(\overline{x})) = \top$ for $1 \leqslant j \leqslant 2$, but $\mathrm{con}(x_3, g(\overline{x})) = \bot$.

[3] To enhance readability we drop parentheses in expressions of the form $x + y \oplus z$, which are to be read as $(x + y) \oplus z$ rather than $x + (y \oplus z)$. Note that these expressions are in general not equivalent, e.g. $(1 + 0) \oplus \omega = \omega + 1$ but $1 + (0 \oplus \omega) = \omega$.

Next, we derive formulas expressing comparisons. Consider ROEs $f(\overline{x})$ and $g(\overline{x})$ as in (4), (5). As a criterion to check whether $[\alpha](f(\overline{x})) > [\alpha](g(\overline{x}))$ for all assignments α, we use the following underapproximation, which is a tradeoff between accuracy and efficiency.

▶ **Definition 15.** Let $f(\overline{x})$ and $g(\overline{x})$ be ROEs as in (4), (5).

$$
\begin{aligned}
[f(\overline{x}) \geqslant g(\overline{x})] &= [f(\overline{x}) \geqslant_0 g(\overline{x})] \wedge \bigwedge_{1 \leqslant i \leqslant n} [f(\overline{x}) \geqslant_i g(\overline{x})] \\
[f(\overline{x}) \geqslant_0 g(\overline{x})] &= ([f'(\overline{x}) >_0 g'(\overline{x})] \wedge f_\omega > 0) \vee ([f'(\overline{x}) \geqslant_0 g'(\overline{x})] \wedge f_\omega \geqslant g_\omega \wedge f_0 \geqslant g_0) \vee \\
&\quad (g_\omega = 0 \wedge f_0 \geqslant g_0) \\
[f(\overline{x}) \geqslant_i g(\overline{x})] &= \neg\mathrm{con}(x_i, g(\overline{x})) \vee && \text{(a)} \\
&\quad ([f'(\overline{x}) \geqslant_i g'(\overline{x})] \wedge f_\omega \geqslant g_\omega \wedge g_i = 0 \wedge \hat{g}_i = 0) \vee && \text{(b)} \\
&\quad (\mathrm{con}(x_i, \omega^{f'(\overline{x})} f_\omega) \wedge \neg\mathrm{con}(x_i, \omega^{g'(\overline{x})} g_\omega)) \vee && \text{(c)} \\
&\quad (\mathrm{con}(x_i, \omega^{f'(\overline{x})} f_\omega) \wedge [f'(\overline{x}) \geqslant_i g'(\overline{x})] \wedge f_\omega > g_\omega) \vee && \text{(d)} \\
&\quad (\mathrm{con}(x_i, \omega^{f'(\overline{x})} f_\omega) \wedge [f'(\overline{x}) \geqslant_i g'(\overline{x})] \wedge f_\omega = g_\omega \wedge \hat{f}_i \geqslant \hat{g}_i) \vee && \text{(e)} \\
&\quad (\neg\mathrm{con}(x_i, \omega^{g'(\overline{x})} g_\omega) \wedge \hat{f}_i \geqslant \hat{g}_i \wedge f_i + \hat{f}_i \geqslant g_i + \hat{g}_i) \vee && \text{(f)} \\
&\quad ((\mathrm{zero}(g'(\overline{x})) \vee g_\omega = 0) \wedge f_i + \hat{f}_i \geqslant g_i + \hat{g}_i) && \text{(g)} \\
[f(\overline{x}) > g(\overline{x})] &= [f(\overline{x}) \geqslant g(\overline{x})] \wedge [f(\overline{x}) >_0 g(\overline{x})] \\
[f(\overline{x}) >_0 g(\overline{x})] &= ([f'(\overline{x}) \geqslant_0 g'(\overline{x})] \wedge f_\omega \geqslant g_\omega \wedge f_0 > g_0) \vee ([f'(\overline{x}) >_0 g'(\overline{x})] \wedge f_\omega > 0)
\end{aligned}
$$

Here $[f(\overline{x}) >_0 g(\overline{x})]$ ($[f(\overline{x}) \geqslant_0 g(\overline{x})]$) encodes that the constant part in $f(\overline{x})$ is greater (or equal) than the constant part in $g(\overline{x})$, whereas $[f(\overline{x}) \geqslant_i g(\overline{x})]$ encodes that the coefficients of the variable x_i in $f(\overline{x})$ are greater than or equal to the respective coefficients in $g(\overline{x})$. Our comparisons are more involved than the absolute positiveness approach [15] because of ordinal arithmetic. We illustrate the different cases in the encoding of $\geqslant_i$ in the following example.

▶ **Example 16.** Case (a) yields $\omega^{x_1+x_2} \geqslant_1 \omega^{x_2}$ while (b) admits $\omega^{x_1}\,2\,3 \geqslant_1 \omega^{x_1}\,3$. From (c) satisfiability of $\omega^{x_1}\,2 \geqslant_1 x_1\,3$ is obtained while $\omega^{x_1}\,2 \geqslant_1 \omega^{x_1}\,1 + x_1\,5$ is due to (d). Case (e) obviously allows $\omega^{x_1}\,2 + x_1\,2 \geqslant_1 \omega^{x_1}\,2 + x_1\,1$ but also $\omega^{x_1} \geqslant_1 x_1\,10 + \omega^{x_1}$. Case (f) implies $x_1\,2 + \omega^{x_2} \oplus x_1\,3 \geqslant_1 x_1\,3 + \omega^{x_2} \oplus x_1\,2$. Note that if ω^{x_2} consumes the preceding $x_1\,2$ ($x_1\,3$) then $\hat{f}_1 \geqslant \hat{g}_1$ must hold. In the other case the test $f_1 + \hat{f}_1 \geqslant g_1 + \hat{g}_1$ is required. Finally, (g) ensures $x_1\,4 + \omega^{x_2} \oplus x_1\,1 \geqslant_1 x_1\,2 \oplus x_1\,3$. If ω^{x_2} consumes $x_1\,4$ then it also dominates $x_1\,2$. In the other case we need the test $f_1 + \hat{f}_1 \geqslant g_1 + \hat{g}_1$.

Clearly, the encoding of $\geqslant$ is only an approximation. E.g., $[\omega^{x_1+1} \geqslant_1 \omega^{x_1}\,2]$ is not satisfiable, despite the fact that $\omega^{x_1+1} > \omega^{x_1}\,2$. However, it is straightforward to extend Definition 15(b) accordingly.

In contrast to e.g. polynomial and matrix interpretations, ROEs are not closed under composition and (standard/natural) addition. Hence we cannot compute an expression corresponding to the interpretation of a term t with respect to an algebra $\mathcal{O}$ either. Instead, we define ROEs $\mu(t)$ and $\nu(t)$ to under- and overapproximate $t_\mathcal{O}$. To this end we present in Definition 17 bounds for the results of ordinal arithmetic operations (based on the algorithms given in [23]) and demonstrate them in Example 18 before Lemma 19 shows their soundness.

▶ **Definition 17.** Let $f(\overline{x})$ and $g(\overline{x})$ be ROEs as in (4), (5).

(a) For $a \in \mathbb{N}$, let $(f \cdot_\mu a)(\overline{x}) = (f \cdot_\nu a)(\overline{x}) = 0$ if $a = 0$ or $f(\overline{x}) = 0$, and otherwise

$$(f \cdot_\mu a)(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i f_i + \omega^{f'(\overline{x})}(f_\omega \cdot a) \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i(\hat{f}_i \cdot a) \oplus (f_0 \cdot a)$$
$$(f \cdot_\nu a)(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i(f_i \cdot a) + \omega^{f'(\overline{x})}(f_\omega \cdot a) \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i(\hat{f}_i \cdot a) \oplus (f_0 \cdot a)$$

(b) Let $(f \oplus_\mu g)(\overline{x}) = (f \oplus_\nu g)(\overline{x}) = g(\overline{x})$ if $f(\overline{x}) = 0$ and $(f \oplus_\mu g)(\overline{x}) = (f \oplus_\nu g)(\overline{x}) = f(\overline{x})$ if $g(\overline{x}) = 0$. Otherwise, let s_i and t_i abbreviate $\neg\mathrm{con}(x_i, \omega^{f'(\overline{x})} f_\omega)$ and $\neg\mathrm{con}(x_i, \omega^{g'(\overline{x})} g_\omega)$ and let

$$(h, h_\omega) = \begin{cases} (f', f_\omega + 1) & \text{if } [\omega^{f'(\overline{x})} f_\omega > \omega^{g'(\overline{x})} g_\omega] \\ (g', g_\omega + 1) & \text{if } [\omega^{g'(\overline{x})} g_\omega > \omega^{f'(\overline{x})} f_\omega] \\ (\mathrm{omax}(f', g'), f_\omega + g_\omega) & \text{otherwise} \end{cases}$$

and $(k, k_\omega) = [\omega^{f'(\overline{x})} f_\omega > \omega^{g'(\overline{x})} g_\omega] \mathbin{?} (f', f_\omega) : (g', g_\omega)$. Here $b \mathbin{?} t : e$ encodes "if b then t else e". Then

$$(f \oplus_\mu g)(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i \max(f_i s_i, g_i t_i) + \omega^{k(\overline{x})} k_\omega \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i(\hat{f}_i + \hat{g}_i) \oplus (f_0 + g_0)$$
$$(f \oplus_\nu g)(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i(f_i s_i + g_i t_i) + \omega^{h(\overline{x})} h_\omega \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i(\hat{f}_i + \hat{g}_i) \oplus (f_0 + g_0)$$

(c) Let $(f +_\mu g)(\overline{x}) = (f +_\nu g)(\overline{x}) = g(\overline{x})$ if $f(\overline{x}) = 0$ and $(f +_\mu g)(\overline{x}) = (f +_\nu g)(\overline{x}) = f(\overline{x})$ if $g(\overline{x}) = 0$. Otherwise, we define lower and upper bounds for $f + g$ by distinguishing different cases using if-then-else expressions:

$$(f +_\mu g)(\overline{x}) = [\omega^{g'(\overline{x})} g_\omega > \omega^{f'(\overline{x})} f_\omega] \mathbin{?} g(\overline{x}) : f(\overline{x})$$
$$(f +_\nu g)(\overline{x}) = ([g'(\overline{x}) > f'(\overline{x})] \wedge g_\omega > 0) \mathbin{?} \phi_1 : \big([\omega^{f'(\overline{x})} f_\omega > \omega^{g'(\overline{x})} g_\omega] \mathbin{?} \phi_2 : (f \oplus_\nu g)(\overline{x})\big)$$

where $c_0 = ([g'(\overline{x}) > 0] \wedge g_\omega > 0) \mathbin{?} g_0 : f_0 + g_0$ and

$$\phi_1 = \sum_{1 \leqslant i \leqslant n} x_i(f_i s_i t_i + \hat{f}_i t_i + g_i t_i) + \omega^{g'(\overline{x})} g_\omega \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i \hat{g}_i \oplus c_0$$
$$\phi_2 = \sum_{1 \leqslant i \leqslant n} x_i f_i s_i + \omega^{f'(\overline{x})}(f_\omega + 1) \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i(\hat{f}_i t_i + g_i t_i + \hat{g}_i) \oplus c_0$$

(d) Definitions (a)–(c) can be used to inductively set lower and upper bounds for the composition $f(\overline{g}) = f(g_1(\overline{x}), \ldots, g_n(\overline{x}))$. We write $\sum^\mu_{1 \leqslant i \leqslant n} h_i$ to abbreviate $h_1 +_\mu \ldots +_\mu h_n$, and use similar shorthands for $\oplus$ and ν. We set

$$f(\overline{g})_\mu(\overline{x}) = \sum_{1 \leqslant i \leqslant n}^{\mu} g_i(\overline{x}) \cdot_\mu f_i +_\mu \omega^{f'(\overline{g})_\mu(\overline{x})} f_\omega \oplus_\mu \bigoplus_{1 \leqslant i \leqslant n}^{\mu} g_i(\overline{x}) \cdot_\mu \hat{f}_i \oplus_\mu f_0$$
$$f(\overline{g})_\nu(\overline{x}) = \sum_{1 \leqslant i \leqslant n}^{\nu} g_i(\overline{x}) \cdot_\nu f_i +_\nu \omega^{f'(\overline{g})_\nu(\overline{x})} f_\omega \oplus_\nu \bigoplus_{1 \leqslant i \leqslant n}^{\nu} g_i(\overline{x}) \cdot_\nu \hat{f}_i \oplus_\nu f_0$$

(e) Let t be a term, and $\mathcal{O}$ be an ROE algebra. By induction on the term structure we define ROEs $\mu_\mathcal{O}(t)$ and $\nu_\mathcal{O}(t)$ such that $\mu_\mathcal{O}(t) = \nu_\mathcal{O}(t) = t$ if $t \in \mathcal{V}$, whereas $\mu_\mathcal{O}(t) = f_\mathcal{O}(\mu_\mathcal{O}(t_1), \ldots, \mu_\mathcal{O}(t_n))_\mu$ and $\nu_\mathcal{O}(t) = f_\mathcal{O}(\nu_\mathcal{O}(t_1), \ldots, \nu_\mathcal{O}(t_n))_\nu$ if $t = f(t_1, \ldots, t_n)$.

The following example illustrates these definitions of upper and lower bounds for ROE arithmetic.

▶ **Example 18.**

(a) Consider the ROE $f(\overline{x}) = x_1 + x_2$. Then $(f \cdot_\mu 2)(\overline{x}) = x_1 + x_2$ and $(f \cdot_\nu 2)(\overline{x}) = x_1 2 + x_2 2$. We clearly have $x_1 + x_2 \leqslant (x_1 + x_2)2 \leqslant x_1 2 + x_2 2$ for all values of x_1, x_2. Note that $(x_1 + x_2)2 \neq x_1 2 + x_2 2$ since $\cdot$ does not right-distribute over $+$, as shown after Definition 2.

(b) Consider the ROEs $f(\overline{x}) = \omega^{x_1+x_2+1} \oplus x_3 + 1$ and $g(\overline{x}) = x_2 + \omega^{x_1} 2 \oplus x_3$. As $\omega^{x_1+x_2+1} > \omega^{x_1} 2$ we have $(k, k_\omega) = (x_1 + x_2 + 1, 1)$ and $(h, h_\omega) = (x_1 + x_2 + 1, 2)$. Thus $(f \oplus_\mu g)(\overline{x}) = \omega^{x_1+x_2+1} \oplus x_3 2 + 1$ and $(f \oplus_\nu g)(\overline{x}) = \omega^{x_1+x_2+1} 2 \oplus x_3 2 + 1$. It is not difficult to see that

$$\omega^{x_1+x_2+1} \oplus x_3 2 + 1 \leqslant (\omega^{x_1+x_2+1} \oplus x_3 + 1) \oplus (x_2 + \omega^{x_1} 2 \oplus x_3) \leqslant \omega^{x_1+x_2+1} 2 \oplus x_3 2 + 1$$

for all values of x_1, x_2, and x_3.

(c) Consider the ROEs $f(\overline{x}) = x_3 + \omega^{x_2} \oplus x_1$ and $g(\overline{x}) = \omega^{x_1+x_2+1} + 1$. We have $(f +_\mu g)(\overline{x}) = g(\overline{x}) = \omega^{x_1+x_2+1} + 1$ and $(f +_\nu g)(\overline{x}) = x_3 + \omega^{x_1+x_2+1} + 1$. Note that the term $\oplus x_1$ in $f(\overline{x})$ disappears as x_1 is considered in the exponent of $g(\overline{x})$. We have

$$\omega^{x_1+x_2+1} + 1 \leqslant (x_3 + \omega^{x_2} \oplus x_1) + (\omega^{x_1+x_2+1} + 1) \leqslant x_3 + \omega^{x_1+x_2+1} + 1$$

for all values of x_1, x_2, and x_3.

(d) For the ROEs $f(\overline{x}) = x_2 + \omega^{x_1+1}$, $g_1(\overline{x}) = \omega^{x_1} \oplus x_2$, and $g_2(\overline{x}) = \omega^{\omega^{x_1} \oplus x_2} \oplus x_3$ we obtain

$$f(\overline{g})_\mu(\overline{x}) = (\omega^{\omega^{x_1} \oplus x_2} \oplus x_3) +_\mu \omega^{\omega^{x_1} \oplus x_2 + 1} = \omega^{\omega^{x_1} \oplus x_2 + 1}$$
$$f(\overline{g})_\nu(\overline{x}) = (\omega^{\omega^{x_1} \oplus x_2} \oplus x_3) +_\nu \omega^{\omega^{x_1} \oplus x_2 + 1} = x_3 + \omega^{\omega^{x_1} \oplus x_2 + 1}$$

(e) Consider the terms $\ell = \bullet\mathsf{f}(\mathsf{c}(x_1, x_2), x_3)$ and $r = \mathsf{h}(\bullet\mathsf{f}(x_1, x_2), \bullet\bullet\mathsf{f}(\mathsf{f}(x_1, x_2), x_3))$ from rule (C2) of $\mathcal{G}$. Let $\mathcal{O}$ be the ordinal part of the ROE algebra defined in the proof of Theorem 13 such that $\mathsf{h}_\mathcal{O}(x_1, x_2) = x_2 + \omega^{x_1+1}$, $\mathsf{c}_\mathcal{O}(x_1, x_2) = \omega^{x_1} \oplus x_2 + 1$, $\bullet_\mathcal{O}(x_1) = x_1$, and $\mathsf{f}_\mathcal{O}(x_1, x_2) = \omega^{x_1} \oplus x_2$. We have $\mu_\mathcal{O}(\ell) = \nu_\mathcal{O}(\ell) = \omega^{\omega^{x_1} \oplus x_2 + 1} \oplus x_3$. It is easy to see that for $r' = \mathsf{f}(\mathsf{f}(x_1, x_2), x_3)$ we have $\mu_\mathcal{O}(r') = \nu_\mathcal{O}(r') = \omega^{\omega^{x_1} \oplus x_2} \oplus x_3$. From the computation in (d) we thus obtain $\nu_\mathcal{O}(r) = x_3 + \omega^{\omega^{x_1} \oplus x_2 + 1}$. Note that $\mu_\mathcal{O}(\ell) \geqslant \nu_\mathcal{O}(r)$ holds: We obviously have $\mu_\mathcal{O}(\ell) \geqslant_0 \nu_\mathcal{O}(r)$, $\mu_\mathcal{O}(\ell) \geqslant_1 \nu_\mathcal{O}(r)$, and $\mu_\mathcal{O}(\ell) \geqslant_2 \nu_\mathcal{O}(r)$ as the two expressions are equal in the relevant parts, and $\mu_\mathcal{O}(\ell) \geqslant_3 \nu_\mathcal{O}(r)$.

We now show that Definition 17 yields valid over- and underapproximations.

▶ **Lemma 19.** *Let $\mathcal{O}$ be an ROE algebra and t be a term. Then $[\alpha](\mu_\mathcal{O}(t)) \leqslant [\alpha]_\mathcal{O}(t) \leqslant [\alpha](\nu_\mathcal{O}(t))$ for all assignments α.*

Proof. We argue that all approximations in Definition 17 constitute valid lower and upper bounds. Let α be an arbitrary assignment.

(a) It is easy to see that $[\alpha](f(\overline{x}) \cdot a) \leqslant [\alpha](f \cdot_\nu a)(\overline{x})$. For any α in CNF as in (1) and $a \in \mathbb{N}_{>0}$, $\alpha a = \omega^{\alpha_1} a_1 a + \omega^{\alpha_2} a_2 + \cdots + \omega^{\alpha_n} a_n$ [23]. Since for any $1 \leqslant i \leqslant n$ we have $\omega^{\alpha_1} a_1 a + \cdots + \omega^{\alpha_n} a_n \geqslant \omega^{\alpha_1} a_1 + \cdots + \omega^{\alpha_i} a_i a + \cdots + \omega^{\alpha_n} a_n$, $(f \cdot_\mu a)(\overline{x})$ constitutes a safe (though modest) lower bound for $f(\overline{x})a$.

(b) We have

$$f(\overline{x}) \oplus g(\overline{x}) = \left(\sum_{1 \leqslant i \leqslant n} x_i f_i + \omega^{f'(\overline{x})} f_\omega \right) \oplus \left(\sum_{1 \leqslant i \leqslant n} x_i g_i + \omega^{g'(\overline{x})} g_\omega \right) \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i(\hat{f}_i + \hat{g}_i) \oplus (f_0 + g_0)$$

Note that the term $x_i f_i$ disappears in $f(\overline{x}) \oplus g(\overline{x})$ if x_i is considered in $\omega^{f'(\overline{x})}$ and $f_\omega > 0$, and the term $x_i g_i$ disappears in $f(\overline{x}) \oplus g(\overline{x})$ if x_i is considered in $\omega^{g'(\overline{x})}$ and $g_\omega > 0$. Hence we may multiply all occurrences of f_i by s_i, and occurrences of g_i by t_i.
We then have $[\alpha](f \oplus_\mu g)(\overline{x}) \leqslant [\alpha](f(\overline{x}) \oplus g(\overline{x}))$ as $(f \oplus_\mu g)(\overline{x})$ underapproximates $\left(\sum_{1 \leqslant i \leqslant n} x_i f_i + \omega^{f'(\overline{x})} f_\omega\right) \oplus \left(\sum_{1 \leqslant i \leqslant n} x_i g_i + \omega^{g'(\overline{x})} g_\omega\right)$ by a coefficient-wise maximum of the respective components in $f(\overline{x})$ and $g(\overline{x})$.
Concerning the upper bound, it is easy to see that $\omega^{f'(\overline{x})} f_\omega \oplus \omega^{g'(\overline{x})} g_\omega \leqslant \omega^{h(\overline{x})} h_\omega$. As the sum of $x_i f_i$ and $x_i g_i$ can be overapproximated by $(f_i s_i + g_i t_i) x_i$ we have $[\alpha](f(\overline{x}) \oplus g(\overline{x})) \leqslant [\alpha](f \oplus_\nu g)(\overline{x})$.

(c) We clearly have $[\alpha](f +_\mu g)(\overline{x}) \leqslant [\alpha](f(\overline{x}) + g(\overline{x}))$.
Concerning the upper bound, assume for a first case $g' > f'$ and $g_\omega > 0$, so $\omega^{f'(\overline{x})} f_\omega + \omega^{g'(\overline{x})} g_\omega = \omega^{g'(\overline{x})} g_\omega$. Note that the term $x_i \hat{f}_i$ disappears in $f(\overline{x}) + g(\overline{x})$ if x_i is contained in $\omega^{g'(\overline{x})}$ and $g_\omega > 0$, the term $g_i x_i$ disappears as well if x_i is contained in $\omega^{g'(\overline{x})}$ and $g_\omega > 0$, and $f_i x_i$ disappears if x_i occurs in $\omega^{f'(\overline{x})}$ and $f_\omega > 0$, or if x_i occurs in $\omega^{g'(\overline{x})}$ and $g_\omega > 0$. Hence for any variable x_i the sum of $x_i f_i$, $x_i \hat{f}_i$, and $x_i g_i$ can be overapproximated by $x_i(f_i s_i t_i + \hat{f}_i t_i + g_i t_i)$. Therefore $[\alpha](f(\overline{x}) + g(\overline{x})) \leqslant [\alpha](f +_\nu g)(\overline{x})$.
Now suppose $[\omega^{f'(\overline{x})} f_\omega > \omega^{g'(\overline{x})} g_\omega]$, so $\omega^{f'(\overline{x})} f_\omega + \omega^{g'(\overline{x})} g_\omega \leqslant \omega^{f'(\overline{x})}(f_\omega + 1)$. The term $\hat{f}_i x_i$ disappears in $f(\overline{x}) + g(\overline{x})$ if x_i is contained in $\omega^{g'(\overline{x})}$ and $g_\omega > 0$, the term $g_i x_i$ disappears as well if x_i is contained in $\omega^{g'(\overline{x})}$ and $g_\omega > 0$. Hence for any variable x_i the sum of $x_i \hat{f}_i$, $x_i g_i$, and $x_i \hat{g}_i$ can be overapproximated by $x_i(\hat{f}_i t_i + g_i t_i + \hat{g}_i)$ such that $[\alpha](f(\overline{x}) + g(\overline{x})) \leqslant [\alpha](f +_\nu g)(\overline{x})$.
Finally, $f(\overline{x}) + g(\overline{x}) \leqslant f(\overline{x}) \oplus g(\overline{x}) \leqslant (f \oplus_\nu g)(\overline{x})$ holds in any case.

(d) By (a)–(c) and weak monotonicity of the ordinal operations $\cdot$, $+$, and $\oplus$.

(e) By induction on the term structure of t, using (d). ◀

All (approximations of) interpretations are weakly monotone. It is easy to encode a criterion for an interpretation to be simple:

$$\mathrm{simple}(f(\overline{x})) = \bigwedge_{1 \leqslant i \leqslant n} \mathrm{con}(x_i, f(\overline{x}))$$

Thus, given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, we assign every $f \in \mathcal{F}$ an abstract ROE $f_\mathcal{O}$ of some depth d. Compatibility of $\mathcal{R}$ with a simple algebra $\mathcal{O}$ is then expressed by

$$\bigwedge_{\ell \to r \in \mathcal{R}} [\mu(\ell) > \nu(r)] \wedge \bigwedge_{f \in \mathcal{F}} \mathrm{simple}(f_\mathcal{O}(\overline{x}))$$

4.2 Implementation

We implemented ordinal interpretations in the termination tool T$_{\text{T}}$T$_2$ [18]. In version 1.09, which is available from the tool's website,[4] ordinal interpretations can be used by executing `./ttt2 -s HYDRA <file>`. Furthermore, the web interface has been updated accordingly. In this section we discuss crucial issues for a successful implementation. Section 4.2.1 shows how to ensure that the lexicographic combination of partial proofs preserves weak monotonicity. Section 4.2.2 deals with the problem of a compatible variable order and Section 4.2.3 is dedicated to efficiency considerations.

[4] `http://cl-informatik.uibk.ac.at/software/ttt2/`

4.2.1 Lexicographic Combination of Interpretations

The termination proof of the TRS $\mathcal{G}$ (Theorem 7) performs a lexicographic combination of algebras into a simple and weakly monotone algebra. The proof can be seen as the lexicographic product of (1) an ordinal interpretation and (2) a linear (polynomial) interpretation and (3) a matrix interpretation of dimension 2. Regarding automation one can either encode the search for the lexicographic combination or search for (partial) proofs and combine them lexicographically. We adopted the latter, although the lexicographic combination of weakly monotone algebras need not be weakly monotone, as shown by the following example.

▶ **Example 20.** Consider the nonterminating TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b}), \mathsf{b} \to \mathsf{a}\}$. For the weakly monotone simple interpretation $\mathsf{f}_{\mathcal{O}}(x) = x + \omega, \mathsf{b}_{\mathcal{O}} = 1, \mathsf{a}_{\mathcal{O}} = 0$ we have $[\mathsf{f}(\mathsf{a})]_{\mathcal{O}} = \omega \geqslant \omega = [\mathsf{f}(\mathsf{b})]_{\mathcal{O}}$ and $[\mathsf{b}]_{\mathcal{O}} = 1 > 0 = [\mathsf{a}]_{\mathcal{O}}$. If we removed the second rule, then the weakly monotone simple interpretation $\mathsf{f}_{\mathcal{N}}(x) = x + 1, \mathsf{a}_{\mathcal{N}} = 1, \mathsf{b}_{\mathcal{N}} = 0$ shows termination of the remaining rule $\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b})$. Note that the lexicographic combination is no longer weakly monotone, i.e., $[\mathsf{b}]_{\mathcal{O}\times\mathcal{N}} = (1,0) >_{\mathsf{lex}} (0,1) = [\mathsf{a}]_{\mathcal{O}\times\mathcal{N}}$ but $[\mathsf{f}(\mathsf{b})]_{\mathcal{O}\times\mathcal{N}} = (\omega,1) \not\geqslant_{\mathsf{lex}} (\omega,2) = [\mathsf{f}(\mathsf{a})]_{\mathcal{O}\times\mathcal{N}}$.

However, we can recover weak monotonicity by interpreting the second component by a constant whenever the first component is only weakly—but not strictly—monotone, i.e., $f_{\mathcal{O}}(x,y) = (x+\omega, c)$ for some $c \in \mathbb{N}$. To achieve this goal in the implementation we consider relative rewriting and add a rule $f' \to f(x_1,\dots,x_n)$ in the relative part whenever $f_{\mathcal{O}}$ is not strictly monotone. Here f' is a fresh constant. In the presence of a rule $f' \to f(x_1,\dots,x_n)$, compatible interpretations satisfy $f_{\mathcal{A}}(x_1,\dots,x_n) = c$ for some c in the domain of $\mathcal{A}$. The idea is demonstrated by the following example.

▶ **Example 21** (Example 20 revisited)**.** Consider the TRS $\mathcal{R}$ from Example 20. After applying the first interpretation we obtain the relative TRS $\{\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b})\}/\{\mathsf{f}' \to \mathsf{f}(x)\}$. Although this system is terminating there is no compatible interpretation since f may not depend on its arguments due to the second rule.

However, adding rules $f' \to f(x_1,\dots,x_n)$ is likely to disable the orientation of rules whose left-hand sides are rooted by f (to satisfy $[\alpha]_{\mathcal{A}}(f') \geqslant [\alpha]_{\mathcal{A}}(f(x_1,\dots,x_n))$ the interpretation of f may not depend on its arguments) and consequently the termination proof might not be successful. To avoid this situation in the implementation we add constraints demanding to orient such rules only if the interpretation of f is not strictly monotone. Then rules rooted with f must be oriented before a rule $f' \to f(x_1,\dots,x_n)$ is added.

Another necessary requirement is that the (lexicographic) algebra is simple. Again we avoid an explicit lexicographic encoding. Rather, in a preprocessing step for every $f \in \mathcal{F}$ we add the embedding rules $f(x_1,\dots,x_n) \to x_i$ (for $1 \leqslant i \leqslant n$) into the relative component of the TRS. This then ensures $[\alpha]_{\mathcal{A}}(f(x_1,\dots,x_n)) \geqslant [\alpha]_{\mathcal{A}}(x_i)$ for each $1 \leqslant i \leqslant n$.

Hence for a TRS $\mathcal{R}$ over a signature $\mathcal{F}$ the procedure amounts to the following three steps:

1. $\mathcal{S} := \{f(x_1,\dots,x_n) \to x_i \mid 1 \leqslant i \leqslant n, f \in \mathcal{F}\}$.
2. Find an algebra $\mathcal{A}$ satisfying $\mathcal{R} \cup \mathcal{S} \subseteq \geqslant_{\mathcal{A}}$ and $\mathcal{R} \cap >_{\mathcal{A}} \neq \varnothing$.

   $\mathsf{Nmon}_{\mathcal{A}}(\mathcal{R}) := \{f' \to f(x_1,\dots,x_n) \mid 1 \leqslant i \leqslant n, f \in \mathcal{F}, f_{\mathcal{A}} \text{ is not strictly monotone}\}$.

   $\mathcal{R} := \mathcal{R} \setminus >_{\mathcal{A}}$ and $\mathcal{S} := (\mathcal{S} \setminus >_{\mathcal{A}}) \cup \mathsf{Nmon}_{\mathcal{A}}(\mathcal{R})$.
3. If $\mathcal{R} = \varnothing$ then output *terminating* else go to step (2).

Instead of proving termination of $\mathcal{R}$ we try to establish termination of $\mathcal{R}$ relative to $\mathcal{S}$ (cf. step (1)). This pre-processing step ensures that the algebras in step (2) are simple. Step (2) employs SMT to find appropriate ROEs and matrix interpretations (of different dimensions), respectively. Note that this step may fail and cause the procedure to abort. Adding $\mathsf{Nmon}_{\mathcal{A}}(\mathcal{R})$ in the relative part ensures that the lexicographic combination of the used algebras is weakly monotone.

4.2.2 Compatible Variable Orders

When interpreting or comparing terms we might get ROEs not having the same variable order. E.g., the rule $\mathsf{s}(\mathsf{g}(x,y)) \to \mathsf{g}(y,x)$ results in the constraint $x+y+1 > y+x$, if $\mathsf{g}_{\mathcal{O}}(x,y) = x+y$ and $\mathsf{s}_{\mathcal{O}}(x) = x+1$. The assignment $\alpha(x) = 1$ and $\alpha(y) = \omega$ yields $1+\omega+1 = \omega+1 \not> \omega+1$ but the encoding $[x+y+1 > y+x]$ is satisfiable. The same effect also happens in arithmetic operations, e.g. the overapproximation of $+$ in Lemma 19(d). Taking $\mathsf{f}_{\mathcal{O}}(x,y) = \mathsf{g}_{\mathcal{O}}(x,y) = x+y$, and $\alpha(x) = 1$, $\alpha(y) = \omega$, the term $\mathsf{f}(\mathsf{g}(x,y),\mathsf{g}(y,x))$ evaluates to $(1+\omega)+(\omega+1) = \omega 2+1$ but the overapproximation based on the variable order $[x,y]$ yields $2+\omega 2 = \omega 2$. Clearly $\omega 2+1 \not\leqslant \omega 2$. Hence we have to add a constraint expressing that two ordinal expressions have *compatible* variable orders (in the standard addition part). Let $\sum_{1\leqslant i\leqslant n} x_i f_i$ and $\sum_{1\leqslant i\leqslant n} y_i g_i$ be ordinal expressions over the same variables (so $\overline{y}$ is a permutation of $\overline{x}$). Let $i < j$. Two variables x_i and x_j are not compatible if there exist i', j' with $1 \leqslant i' < j' \leqslant n$ such that $x_i = y'_j$ and $x_j = y'_i$. In such a case we constrain one of the coefficients to be zero, i.e., $f_i = 0 \vee f_j = 0 \vee g_{i'} = 0 \vee g_{j'} = 0$. For example consider $e_1 = x_1 \cdot 1 + x_2 \cdot 1$, $e_2 = x_2 \cdot 1 + x_1 \cdot 1$, and $e_3 = x_2 \cdot 1 + x_1 \cdot 0$. Then e_1 and e_2 do not have compatible variable orders while e_1 and e_3 do have.

4.2.3 Efficiency

While the implementation fixes some initial depth d for the interpretation of function symbols, this depth increases when evaluating terms (when composing $f(g_1(\overline{x}),\ldots,g_n(\overline{x}))$). It turned out that for efficiency it is necessary to bound the depth of expressions occurring in evaluations of terms. Dropping parts of an interpretation is sound as an underapproximation while for the overapproximation we add constraints (to the SMT solver) that the dropped part must evaluate to zero.

For the automatic termination proof of the TRS $\mathcal{G}$ in $\mathsf{TTT2}$ we (lexicographically) combine ordinal interpretations with matrix interpretations [9]. Then, $\mathsf{TTT2}$ manages $\mathcal{G}$ within six seconds when using depth 1 for interpreting function symbols and limiting the depth of evaluations to 2. The CNF of the underlying SAT problem has approximately 86.000 variables and 217.000 clauses.

5 Hydra Battles

In their influential paper [17], Kirby and Paris also presented the battle of Hercules and Hydra as a combinatorial game on trees. Generalizations of the Hydra battle are found in many papers (Fleischer [10] contains a nice survey) and several different encodings of the battle into a termination problem of a specific TRS can be found in the literature [4, 6, 8, 7, 20, 28]. Not all of these TRSs faithfully model the battle, and termination of some of them are not independent of Peano arithmetic.

▶ **Example 22.** Touzet [28] presents the following TRS $\mathcal{H}$ to describe the battle between Hercules and Hydra for starting terms corresponding to ordinals $\alpha < \omega^{\omega^\omega}$ and using a

so-called *standard strategy*:

$$\begin{array}{lll}
\circ x \to \bullet [] x & \mathsf{H}(0,x) \to \circ x & \bullet \mathsf{c}^1(x,y) \to \mathsf{c}^1(x,\mathsf{H}(x,y)) \\
\bullet [] x \to [] \bullet \bullet x & \bullet \mathsf{H}(\mathsf{H}(0,y),z) \to \mathsf{c}^1(y,z) & \bullet \mathsf{c}^2(x,y,z) \to \mathsf{c}^2(x,\mathsf{H}(x,y),z) \\
[] \circ x \to \circ [] x & \bullet \mathsf{H}(\mathsf{H}(\mathsf{H}(0,x),y),z) \to \mathsf{c}^2(x,y,z) & \mathsf{c}^1(y,z) \to \circ z \\
\bullet x \to x & \mathsf{c}^2(x,y,z) \to \circ \mathsf{H}(y,z) &
\end{array}$$

So far all termination tools failed on this example whose derivational complexity cannot be bounded by a multiple recursive function. Its termination can be shown by the following simple and weakly monotone interpretation $\mathcal{A}$ over the domain $\mathbb{O} \times \mathbb{N} \times \mathbb{N}$, where $f(x,y) = y + \omega^{x+1}$ [28]:

$$\begin{aligned}
\mathsf{0}_{\mathcal{A}} &= (0,0,0) & []_{\mathcal{A}}(x,m,n) &= (x,2m+2,n) \\
\mathsf{H}_{\mathcal{A}}((x,m,n),(y,k,l)) &= (\omega^x \oplus y,0,0) & \circ_{\mathcal{A}}(x,m,n) &= (x,2m+3,n) \\
\mathsf{c}^1_{\mathcal{A}}((x,m,n),(y,k,l)) &= (f(x,y),0,0) & \bullet_{\mathcal{A}}(x,m,n) &= (x,m,n+m+1) \\
\mathsf{c}^2_{\mathcal{A}}((x,m,n),(y,k,l),(z,i,j)) &= (\omega^{f(x,y)} \oplus z,0,0) & &
\end{aligned}$$

Compared to $\mathcal{G}$, TTT2 requires more resources (initial depth 2, intermediate depth 3, 12 seconds, 117.000 variables, 300.000 clauses) to automatically prove termination of $\mathcal{H}$. This is surprising as the derivational complexity of $\mathcal{G}$ far exceeds that of the Hydra system $\mathcal{H}$, which is bounded by the Hardy function $H_{\omega^{\omega^\omega}}$.

In [2], Beklemishev presents two infinite and one finite TRS $\mathcal{W}$ describing the Worm battle (corresponding to a one-dimensional version of Buchholz' Hydra game [3], first introduced by Hamano and Okada [13]). The finite system $\mathcal{W}$ consists of the following rules:

$$\begin{array}{lll}
(x \cdot y) \cdot z \to x \cdot (y \cdot z) & \mathsf{a}(\mathsf{f}(x)) \to \mathsf{f}(\mathsf{a}(x)) & \mathsf{a}(x \cdot y) \to \mathsf{a}(x) \cdot y \\
\mathsf{a}(\mathsf{b}_1(x)) \to \mathsf{b}_1(\mathsf{a}(x)) & \mathsf{f}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{f}(x)) & \mathsf{b}(x) \cdot y \to \mathsf{b}(x \cdot y) \\
\mathsf{a}(\mathsf{f}(0 \cdot x)) \to \mathsf{b}_1((\mathsf{f}(0 \cdot x)) \cdot (0 \cdot \mathsf{f}(x))) & \mathsf{a}(\mathsf{f}(0)) \to \mathsf{b}_1(\mathsf{f}(0) \cdot 0) & \mathsf{b}_1(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{b}(x)) \\
\mathsf{f}(0 \cdot x) \to \mathsf{b}(0 \cdot \mathsf{f}(x)) & \mathsf{f}(0) \to \mathsf{b}(0) & \mathsf{c}(\mathsf{b}(x)) \to \mathsf{c}(\mathsf{a}(x)) \\
\mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{a}(x)) & &
\end{array}$$

Termination of $\mathcal{W}$ is proved in [2] by relating it to another, infinite TRS. We have not been able to prove termination using ordinal interpretations, a goal we mention as a future aim.

Needless to say, there will always be TRSs whose termination is out of reach of automatic tools. With our implementation of ordinal interpretations, one cannot prove termination of TRSs whose derivational complexity goes beyond ϵ_0. Lepper [20] presented an infinite sequence $(\mathcal{R}_k)_{k \geqslant 1}$ of TRSs that simulate Hydra battles. The derivational complexity Δ_k of $\mathcal{R}_k$ approaches the small Veblen ordinal $\vartheta(\Omega^\omega)$ when k tends to infinity.

Furthermore, as our implementation is based on Theorem 5 it cannot cope with TRSs that are non-simply terminating. Hence the very first encoding of the Hydra battle in [7] defeats TTT2. A (difficult) termination proof of this TRS can be found in Moser [24]. While dependency pairs [1] go beyond simple termination they do not solve the intrinsic problems of this work (exceeding multiple recursion, establishing weak monotonicity) and have hence been discarded for ease of presentation.

6 Conclusion

6.1 Summary

We have encoded Goodstein's sequence as a TRS and discussed automation of a termination criterion which can cope with this system. Furthermore our implementation is also successful on an encoding of the battle of Hercules and Hydra, for which a (sound) automatic termination proof has been lacking so far. While preliminary experiments on the termination problems database TPDB (see footnote 1 on page 335) did not yield proofs for previously unknown problems, we regard the main attraction of our method that it allows to go beyond multiple recursive derivation length. As shown in the paper, automation of lexicographic combinations of termination proofs with respect to Theorem 5 is more challenging than in the standard setting where strictly monotone algebras are employed.

6.2 Future Work

Concerning future work we want to improve the approximations of our term interpretation encodings. Here we discuss scalar multiplication. Since the approximations must be correct for all values of $\overline{x}$, the overapproximation $(f \cdot_\nu a)(\overline{x})$ is already optimal. To see this consider $(x + y) \cdot_\nu 2$ for natural values of x and y. Inspecting the proof of Lemma 19(a), instead of the current underapproximation $(f \cdot_\mu a)(\overline{x})$ we could also use (if $a > 0$):

$$(f \cdot_{\mu'} a)(\overline{x}) = \sum_{1 \leqslant i \leqslant n} x_i(f_i \cdot e_i) + \omega^{f'(\overline{x})}(f_\omega \cdot a) \oplus \bigoplus_{1 \leqslant i \leqslant n} x_i\hat{(}f_i \cdot a) \oplus (f_0 \cdot a)$$

where exactly one of e_i is a and all others are one. The underlying SMT solver can then choose an appropriate summand to be multiplied with a such that subsequent operations (addition, comparison, etc.) benefit. Refining the approximations for other operations (addition/comparison) is more involved and it is unclear if the additional precision is in a suitable ratio with the increasing difficulty of the resulting SMT problems.

Furthermore we stress that (efficient) approximations (similar to the ones presented) will also be necessary for a successful implementation of e.g. elementary functions as proposed by Lescanne [21]. Despite the recent efforts of Lucas [22], to our knowledge an implementation of such interpretations is still lacking. We anticipate that automation of elementary functions might give new automatic termination proofs for problems from TPDB.

Acknowledgments. We thank Georg Moser for his comments on Hydra battles and ordinals, René Thiemann for communicating Example 20, and the anonymous reviewers for many helpful remarks.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1-2):133–178, 2000.

2 L. Beklemishev. Representing Worms as term rewriting systems. In *Proc. Mini-Workshop: Logic, Combinatorics and Independence Results*, volume 3(4) of *Oberwolfach Reports*, pages 3093–3095. European Mathematical Society, 2006.

3 W. Buchholz. An independence result for $(\pi^1_1 - \mathrm{CA}) + \mathrm{BI}$. *Ann. Pure Appl. Logic*, 33:131–155, 1987.

4 W. Buchholz. Another rewrite system for the standard Hydra battle. In *Proc. Mini-Workshop: Logic, Combinatorics and Independence Results*, volume 3(4) of *Oberwolfach Reports*, pages 3099–3102. European Mathematical Society, 2006.

5 E.A. Cichon. A short proof of two recently discovered independence results using recursion theoretic methods. *Proc. Am. Math. Soc.*, 87(4):704–706, 1983.

6 N. Dershowitz. Trees, ordinals, and termination. In *TAPSOFT*, volume 668 of *LNCS*, pages 243–250, 1993.

7 N. Dershowitz and J.-P. Jouannaud. *Rewrite Systems. Handbook of Theoretical Computer Science*, volume B: Formal Models and Sematics, pages 243–320. Elsevier, 1990.

8 N. Dershowitz and G. Moser. The Hydra battle revisited. In *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *LNCS*, pages 1–27, 2007.

9 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(2-3):195–220, 2008.

10 R. Fleischer. Die another day. *Theory of Computing Systems*, 44(2):205–214, 2009.

11 G. Gentzen. Die widerspruchsfreiheit der reinen zahlentheorie. *Mathematische Annalen*, 122:493–565, 1936.

12 R. L. Goodstein. On the restricted ordinal theorem. *J. Symb. Log.*, 9(2):33–41, 1944.

13 M. Hamano and M. Okada. A relationship among Gentzen's proof-reduction, Kirby-Paris' Hydra game and Buchholz's Hydra game. *Math. Logic Quart.*, 43:103–120, 1997.

14 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations (preliminary version). In *RTA*, volume 355 of *LNCS*, pages 167–177, 1989.

15 H. Hong and D. Jakuš. Testing positiveness of polynomials. *JAR*, 21(1):23–38, 1998.

16 W. Just and M. Weese. *Discovering Modern Set Theory. I The Basics*. American Mathematical Society, 1996.

17 L. Kirby and J. Paris. Accessible independency results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:285–325, 1982.

18 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *RTA*, volume 5595 of *LNCS*, pages 295–304, 2009.

19 I. Lepper. Derivation lengths and order types of Knuth-Bendix orders. *TCS*, 269(1-2):433–450, 2001.

20 I. Lepper. Simply terminating rewrite systems with long derivations. *Archive for Mathematical Logic*, 43(1):1–18, 2004.

21 P. Lescanne. Termination of rewrite systems by elementary interpretations. *Formal Asp. Comp.*, 7(1):77–90, 1995.

22 S. Lucas. Automatic proofs of termination with elementary interpretations. *ENTCS*, 258(1):41–61, 2009.

23 P. Manolios and D. Vroon. Ordinal arithmetic: Algorithms and mechanization. *JAR*, 34(4):387–423, 2005.

24 G. Moser. The Hydra battle and Cichon's principle. *AAECC*, 20(2):133–158, 2009.

25 G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. *LMCS*, 7(3), 2011.

26 A. Schnabl. *Derivational Complexity Analysis Revisited*. PhD thesis, University of Innsbruck, 2012.

27 TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

28 H. Touzet. Encoding the Hydra battle as a rewrite system. In *MFCS*, volume 1450 of *LNCS*, pages 267–276, 1998.

29 A. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–68, 1949.

30 A. Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *TCS*, 139(1-2):355–362, 1995.

31 S. Winkler, H. Zankl, and A. Middeldorp. Ordinals and Knuth-Bendix orders. In *LPAR*, volume 7180 of *LNCS (ARCoSS)*, pages 420–434, 2012.

32 H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *LPAR*, volume 6355 of *LNCS (LNAI)*, pages 481–500, 2010.

33 H. Zankl, S. Winkler, and A. Middeldorp. Automating ordinal interpretations. In *WST*, pages 94–98, 2012.

34 H. Zantema. The termination hierarchy for term rewriting. *AAECC*, 12(1-2):3–19, 2001.

Confluence by Decreasing Diagrams – Formalized

Harald Zankl

Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria
harald.zankl@uibk.ac.at

Abstract

This paper presents a formalization of decreasing diagrams in the theorem prover Isabelle. It discusses mechanical proofs showing that any locally decreasing abstract rewrite system is confluent. The valley and the conversion version of decreasing diagrams are considered.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases term rewriting, confluence, decreasing diagrams, formalization

Digital Object Identifier 10.4230/LIPIcs.RTA.2013.352

1 Introduction

Formalizing confluence criteria has a long history in λ-calculus. Huet [8] proved a stronger variant of the parallel moves lemma in Coq. Isabelle/HOL was used in [11] to prove the Church-Rosser property of β, η, and $\beta\eta$. For β-reduction the standard Tait/Martin-Löf proof as well as Takahashi's proof [23] were formalized. The first mechanically verified proof of the Church-Rosser property of β-reduction was done using the Boyer-Moore theorem prover [20]. The formalization in Twelf [18] was used to formalize the confluence proof of a specific higher-order rewrite system in [22].

Newman's lemma (for abstract rewrite systems) and Knuth and Bendix' critical pair theorem (for first-order rewrite systems) have been proved in [19] using ACL. An alternative proof of the latter in PVS, following the higher-order structure of Huet's proof, is presented in [7]. PVS is also used in the formalization of the lemmas of Newman and Yokouchi in [6]. Knuth and Bendix' criterion has also been formalized in Coq [3] and Isabelle/HOL [25].

Decreasing diagrams [13] are a complete characterization of confluence for abstract rewrite systems whose convertibility classes are countable. As a criterion for abstract rewrite systems, they can easily be applied for first- and higher-order rewriting, including term rewriting and the λ-calculus. Furthermore, decreasing diagrams yield constructive proofs of confluence [16] (in the sense that the joining sequences can be computed based on the divergence). We are not aware of a (complete) formalization of decreasing diagrams in any theorem prover (see remarks in Section 6).

In this paper we discuss a formalization of decreasing diagrams in the theorem prover Isabelle/HOL. (In the sequel we just call it Isabelle.) We closely follow the proofs in [13,15]. For alternative proofs see [1,10] or [5,9,17] where proof orders play an essential role. The main contributions of this paper are (two) mechanical proofs of Theorem 1 in Isabelle.

▶ **Theorem 1** ([13,15]). *A locally decreasing abstract rewrite system is confluent.* ◀

As a consequence all definitions (lemmata) in this paper have been formalized (proved) in Isabelle. The definitions from the paper are (modulo notation) identical to the ones used in Isabelle. Our formalization (Decreasing_Diagrams.thy, available from [27]) consists of approximately 1600 lines of Isabelle code in the Isar style and contains 31 definitions

24th International Conference on Rewriting Techniques and Applications (RTA'13).
Editor: Femke van Raamsdonk; pp. 352–367

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


Table 1 Predefined Isabelle operators.

meaning	set	multiset	sequence/list	[13]
empty	$\{\}$	$\{\#\}$	$[]$	$\emptyset/\epsilon$
singleton	$\{\alpha\}$	$\{\#\alpha\#\}$	$[\alpha]$	$\{\alpha\}/[\alpha]/\alpha$
membership	$\alpha \in S$	$\alpha \in\# M$	–	$\in$
union/concatenation	$S \cup T$	$M + N$	$\sigma@\tau$	$\uplus/\sigma\tau$
intersection	$S \cap T$	$M \#\cap N$	–	$\cap$
difference	$S - T$	$M - N$	–	$-$
sub(multi)set	$S \subseteq T$	$M \leq N$	–	$\subseteq$

and 122 lemmata. The valley version [13] amounts to ca. 1000 lines, 22 definitions, and 97 lemmata while the conversion version [15] has additional 600 lines of Isabelle comprising 9 definitions and 25 lemmata. Our formalization imports the theory Multiset.thy from the Isabelle library and Abstract_Rewriting.thy [21] from the Archive of Formal Proofs. We used Isabelle 2012 and the Archive of Formal Proofs from July 30, 2012.

The remainder of this paper is organized as follows. In the next section we recall helpful preliminaries for our formalization of [13], which is described in Section 3. The conversion version of decreasing diagrams [15] is the topic of Section 4. In Section 5 we highlight changes to (and omissions in) the proofs from [13, 15] before we conclude in Section 6.

2 Preliminaries

We assume familiarity with rewriting [24] and decreasing diagrams [13]. Basic knowledge of Isabelle [12] is not essential but may be helpful.

Given a relation $\to$ we write $\leftarrow$ for its inverse, $\twoheadrightarrow$ for its transitive closure, and $\to^=$ (in pictures also $\xrightarrow{=}$) for its reflexive closure. We write $\leftrightarrow$ for $\to$ or $\leftarrow$ and denote sets by S, T, U, multisets by M, N, I, J, K, Q, single labels by α, β, and γ, and lists of labels by σ, τ, υ, κ, μ, and ρ (possibly primed or indexed).

Table 1 gives an overview of several predefined operators in Isabelle for sets, multisets, and lists (sequences) where we also incorporated the notation from [13] in the rightmost column. In the paper we will use the Isabelle notation, but drop the @ for concatenating sequences and write α instead of $[\alpha]$. In addition to the operators provided by Isabelle, we need the difference (intersection) of a multiset with a set. Here $M -s S$ ($M \cap s S$) removes (keeps) all occurrences of elements in M that are in S. Sometimes it will be necessary to convert e.g. a multiset to a set (or a list). In the paper we leave these conversions implicit, since no confusion can arise. We establish the following useful equivalences:

▶ **Lemma 2** (parts of [13, Lemma A.3])**.**

1. $(M + N) -s S = (M -s S) + (N -s S)$
2. $(M -s S) -s T = M -s (S \cup T)$
3. $M = (M \cap s S) + (M -s S)$
4. $(M -s T) \cap s S = (M \cap s S) -s T$

Proof. By unfolding the definitions of multiset and the operators. ◀

3 Formalization of Decreasing Diagrams

We assume familiarity with the original proof of decreasing diagrams in [13], upon which our formalization in this section is based. Nevertheless we will recall the important definitions and lemmata. However, we only give proofs if our proof deviates from the original argument. In addition we state (sometimes small) key results, since an effective collection of lemmata is crucial for completely formal proofs.

The remainder of this section is organized as follows: Section 3.1 describes our results on multisets. Section 3.2 is dedicated to decreasingness (of sequences of labels) and Section 3.3 is concerned with an alternative formulation of local decreasingness. Afterwards, Section 3.4 lifts decreasingness (from labels) to diagrams. Well-foundedness of the measure (on peaks) is proved in Section 3.5, where we also establish the main result.

3.1 Multisets

In the sequel we assume $\prec$ to be a transitive and irreflexive binary relation.

▶ **Definition 3** ([13, Definition 2.5])**.**

1. The set $\Upsilon\alpha$ is the strict order ideal generated by (or *down-set* of) α, defined by $\Upsilon\alpha = \{\beta \mid \beta \prec \alpha\}$. This is extended to sets $\Upsilon S = \bigcup_{\alpha\in S} \Upsilon\alpha$. We define ΥM and $\Upsilon\sigma$ to be the down-set generated by the set of elements in M and σ, respectively.
2. The *(standard) multiset extension* (denoted by $\prec_{\mathsf{mul}}$) of $\prec$ is defined by

$$M \prec_{\mathsf{mul}} N \text{ if } \exists\, I\ J\ K.\ M = I + K,\ N = I + J,\ K \subseteq \Upsilon J, \text{ and } J \neq \{\#\}$$

The relation $\preccurlyeq_{\mathsf{mul}}$ is obtained by removing the last condition ($J \neq \{\#\}$). Note that $\preccurlyeq_{\mathsf{mul}}$ is the reflexive closure of $\prec_{\mathsf{mul}}$ (cf. Lemma 39 in Section 5).

The following result is not mentioned in [13]—while [14, Proposition 1.4.8(3)] shows a more general result—but turned out handy for our formalization.

▶ **Lemma 4.** $\Upsilon(\Upsilon S) \subseteq \Upsilon S$

Proof. Assume $x \in \Upsilon(\Upsilon S)$. By Definition 3 there must be a $y \in \Upsilon S$ with $x \prec y$. From $y \in \Upsilon S$ we obtain a $z \in S$ with $y \prec z$. Then $x \prec z$ by transitivity of $\prec$ and hence $x \in \Upsilon S$. ◀

The multiset extension inherits some properties of the base relation, which we will implicitly use in the sequel.

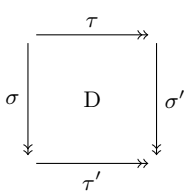
▶ **Lemma 5.** *Let $\prec$ be a transitive and well-founded relation. Then $\prec_{mul}$ is transitive and well-founded, and $\preccurlyeq_{mul}$ is reflexive and transitive.*

Proof. By Lemmata 38 and 39 in combination with existing results in Multiset.thy. ◀

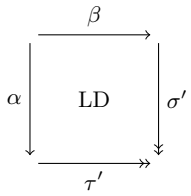
We can now establish the following properties.

▶ **Lemma 6** ([13, Lemma 2.6])**.**

1. $\Upsilon(S \cup T) = \Upsilon S \cup \Upsilon T$ *and* $\Upsilon(\sigma\tau) = \Upsilon\sigma \cup \Upsilon\tau$ *and* $\Upsilon(M -s\ S) \supseteq \Upsilon M -s\ \Upsilon S$
2. $M \leq N \Rightarrow M \preccurlyeq_{mul} N \Rightarrow \Upsilon M \subseteq \Upsilon N$
3. $M \preccurlyeq_{mul} N \Rightarrow \exists\, I\ J\ K.\ M = I + K \wedge N = I + J \wedge K \subseteq \Upsilon J \wedge J \#\cap K = \{\#\}$
4. $N \neq \{\#\} \wedge M \subseteq \Upsilon N \Rightarrow M \prec_{mul} N$
5. $M \preccurlyeq_{mul} N \Rightarrow M -s\ \Upsilon S \preccurlyeq_{mul} N -s\ \Upsilon S$
6. $M \preccurlyeq_{mul} N \Leftrightarrow Q + M \preccurlyeq_{mul} Q + N$
7. $Q \subseteq \Upsilon N - \Upsilon M \wedge M \preccurlyeq_{mul} N \Rightarrow Q + M \preccurlyeq_{mul} N$
8. $S \subseteq T \Rightarrow M -s\ T \preccurlyeq_{mul} M -s\ S$
9. $M \prec_{mul} N \Rightarrow Q + M \prec_{mul} Q + N$



(a) Decreasing diagram.



(b) Locally decreasing diagram.

Figure 1 Diagrams.

Note that statements (5) and (6) slightly differ from [13, Lemma 2.6](5,6), but are easier to apply. The (easy) the statements of (8) and (9) are not mentioned in [13], which we required for [13, Lemmata 3.5 and 3.6].

3.2 Decreasingness

We define the *lexicographic maximum* measure, which maps lists to multisets, inductively.

▶ **Definition 7** ([13, Definition 3.2])**.**

- $|[]| = \{\#\}$
- $|\alpha\sigma| = \{\#\alpha\#\} + (|\sigma| -_s \Upsilon\alpha)$

The next lemma establishes properties of the lexicographic maximum measure.

▶ **Lemma 8** ([13, Lemma 3.2])**.**

1. $\Upsilon|\sigma| = \Upsilon\sigma$
2. $\Upsilon|\sigma\tau| = |\sigma| + (|\tau| -_s \Upsilon\sigma)$

Proof.

1. By induction on σ. The base case is trivial. Using Lemma 6(1) the inductive step amounts to $\Upsilon\alpha \cup \Upsilon(|\sigma| -_s \Upsilon\alpha) = \Upsilon\alpha \cup \Upsilon\sigma$. The inclusion from left to right follows from the induction hypothesis. For the inclusion from right to left we proceed by case analysis. If $x \in \Upsilon\alpha$ then the result immediately follows. If $x \notin \Upsilon\alpha$ then $x \in \Upsilon\sigma$ and from the induction hypothesis $x \in \Upsilon|\sigma|$. Furthermore $x \notin \Upsilon\alpha$ using Lemma 4 also yields $x \notin \Upsilon(\Upsilon\alpha)$. Hence $x \in \Upsilon|\sigma| -_s \Upsilon(\Upsilon\alpha)$ and from Lemma 6(1) we obtain $x \in \Upsilon(|\sigma| -_s \Upsilon\alpha)$, from which the result follows.
2. By induction on σ, see [13]. ◀

Decreasingness is defined on quadruples (of sequences of labels).

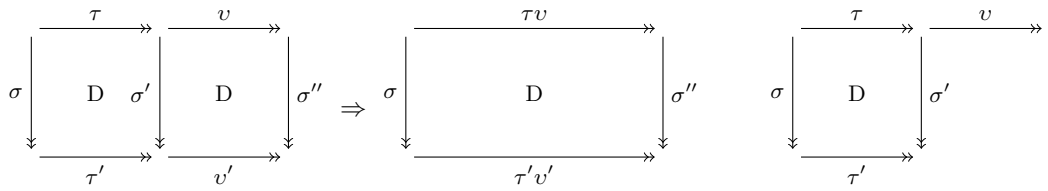
▶ **Definition 9** ([13, Definition 3.3] for labels)**.** The quadruple of labels $(\tau, \sigma, \sigma', \tau')$ is *decreasing* (D) if $|\sigma\tau'| \preccurlyeq_{\mathsf{mul}} |\tau| + |\sigma|$ and $|\tau\sigma'| \preccurlyeq_{\mathsf{mul}} |\tau| + |\sigma|$. For a visualization see Figure 1a.[1]

We write D into a diagram to indicate that its labels are decreasing.
Decreasingness can also be stated differently.

▶ **Lemma 10** ([13, Definition 3.3])**.** *The following two statements are equivalent:*

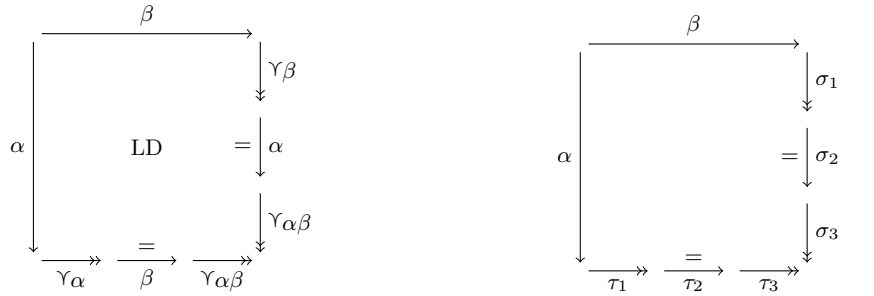
1. $|\sigma\tau'| \preccurlyeq_{mul} |\tau| + |\sigma|$ *and* $|\tau\sigma'| \preccurlyeq_{mul} |\tau| + |\sigma|$
2. $|\tau'| -_s \Upsilon\sigma \preccurlyeq_{mul} |\tau|$ *and* $|\sigma'| -_s \Upsilon\tau \preccurlyeq_{mul} |\sigma|$

[1] Although the results in Sections 3.2 and 3.3 are on labels only for visualization we already use diagrams.



(a) Lemma 11. **(b)** Lemma 12.

Figure 2 Pasting preserves decreasingness and is hypothesis decreasing.



(a) Alternative formulation of local decreasingness. **(b)** Giving names to the joining sequences.

Figure 3 Local diagrams.

Proof. By Lemma 8(2) and Lemma 6(6). ◀

We have followed the (involved) proofs in [13] that pasting preserves decreasingness (Lemma 11) and that pasting is hypothesis decreasing (Lemma 12) without big changes.

▶ **Lemma 11** ([13, Lemma 3.5] for labels)**.** *If* $(\tau, \sigma, \sigma', \tau')$ *and* $(\upsilon, \sigma', \sigma'', \upsilon')$ *are decreasing, then* $(\tau\upsilon, \sigma, \sigma'', \tau'\upsilon')$ *is decreasing (see Figure 2a).*

Proof. As in [13] but we show $(|\upsilon'| -_s \Upsilon\sigma\tau') -_s \Upsilon\tau \preccurlyeq_{\mathsf{mul}} (|\upsilon'| -_s \Upsilon\sigma') -_s \Upsilon\tau$ (instead of $\subseteq$) where we needed Lemma 6(8) (in the last sequence in [13, Proof of Lemma 3.5]). ◀

▶ **Lemma 12** ([13, Lemma 3.6] for labels)**.** *If* τ *is non-empty and we have that* $(\tau, \sigma, \sigma', \tau')$ *is decreasing (see Figure 2b) then* $|\sigma'| + |\upsilon| \prec_{\mathsf{mul}} |\sigma| + |\tau\upsilon|$.

Proof. As in [13] using Lemma 6(9) in the second step. ◀

3.3 Local Decreasingness

Labels $(\beta, \alpha, \sigma', \tau')$ are *locally decreasing* (LD) if they are decreasing and both α and β consist of exactly one label (see Figure 1b). Now, LD can also be formulated differently:

▶ **Lemma 13** ([13, Prop. 3.4])**.** *The form of locally decreasing labels is specified in Figure 3a.*

To show Lemma 13 we give names to the joining sequences as in Figure 3b. Then the condition of Figure 3a can be expressed as:[2]

$$\begin{aligned}\mathrm{LD}' := {} & \sigma_1 \subseteq \Upsilon\beta \wedge \mathsf{length}\ \sigma_2 \leq 1 \wedge \sigma_2 \subseteq \{\alpha\} \wedge \sigma_3 \subseteq \Upsilon\alpha\beta \wedge \\ & \tau_1 \subseteq \Upsilon\alpha \wedge \mathsf{length}\ \tau_2 \leq 1 \wedge \tau_2 \subseteq \{\beta\} \wedge \tau_3 \subseteq \Upsilon\alpha\beta\end{aligned}$$

Local decreasingness of the labels in the diagram of Figure 3a (using Lemma 10) yields

$$\mathrm{LD} := |\sigma'| -s \Upsilon\beta \preccurlyeq_{\mathsf{mul}} |\alpha| \wedge |\tau'| -s \Upsilon\alpha \preccurlyeq_{\mathsf{mul}} |\beta|$$

Hence Lemma 13 states that LD' if and only if LD. This means that

(i) if a local diagram satisfies the conditions in Figure 3a, i.e. LD', then it is decreasing and

(ii) local decreasingness implies that the joining sequences τ' and σ' in Figure 1b can be decomposed into $\tau_1\tau_2\tau_3$ and $\sigma_1\sigma_2\sigma_3$ such that the properties of the local diagram in Figure 3a, i.e. LD', are satisfied.

Lemma 15 will be the key result for (i), but first we establish a useful lemma.

▶ **Lemma 14.** $|\sigma| \leq \sigma$

Proof. By induction on σ. The base case is trivial. The step case amounts to

$$|\alpha\sigma| \;=\; \{\#\alpha\#\} + (|\sigma| -s \Upsilon\alpha) \;\leq\; \{\#\alpha\#\} + (\sigma -s \Upsilon\alpha) \;\leq\; \alpha\sigma$$

using Definition 7 in the first step and the induction hypothesis in the second step. ◀

In the sequel we will view $|\sigma|$ and σ as sets and use $|\sigma| \subseteq \sigma$. Now we can prove the following key result to establish (i).

▶ **Lemma 15.** $\sigma_1 \subseteq \Upsilon\beta \wedge \mathit{length}\ \sigma_2 \leq 1 \wedge \sigma_2 \subseteq \{\alpha\} \wedge \sigma_3 \subseteq \Upsilon\alpha\beta \Rightarrow |\sigma_1\sigma_2\sigma_3| -s \Upsilon\beta \preccurlyeq_{\mathit{mul}} |\alpha|$

Proof. We show

$$(|\sigma_1| -s \Upsilon\beta) + ((|\sigma_2| -s \Upsilon\sigma_1) -s \Upsilon\beta) + (((|\sigma_3| -s \Upsilon\sigma_2) -s \Upsilon\sigma_1) -s \Upsilon\beta) \preccurlyeq_{\mathsf{mul}} \{\#\alpha\#\} \quad (\star)$$

which is equivalent to the conclusion by Lemmata 8(2), 2(1) and Definition 7. The hypothesis contains $\sigma_1 \subseteq \Upsilon\beta$, which together with Lemma 14 yields $|\sigma_1| \subseteq \Upsilon\beta$ and hence

$$|\sigma_1| -s \Upsilon\beta = \{\#\} \tag{1}$$

Similarly from $\sigma_3 \subseteq \Upsilon\alpha\beta$ we get $|\sigma_3| -s (\Upsilon\alpha \cup \Upsilon\beta) = \{\#\}$ and hence

$$|\sigma_3| -s (\Upsilon\sigma_2 \cup \Upsilon\sigma_1 \cup \Upsilon\alpha \cup \Upsilon\beta) = \{\#\} \tag{3}$$

Using $\mathsf{length}\ \sigma_2 \leq 1 \wedge \sigma_2 \subseteq \{\alpha\}$ from the hypothesis we have two cases to consider for σ_2.

- If $\sigma_2 = [\,]$ then

  $$(|\sigma_2| -s \Upsilon\sigma_1) -s \Upsilon\beta = \{\#\} \tag{2}$$

  and from (3) we have

  $$((|\sigma_3| -s \Upsilon\sigma_2) -s \Upsilon\sigma_1) -s \Upsilon\beta \preccurlyeq_{\mathsf{mul}} \{\#\alpha\#\} \tag{3'}$$

  using Lemma 2(2). Then $(\star)$ follows immediately from (1), (2), and (3').

[2] Here length computes the length of a list.

- If $\sigma_2 = [\alpha]$ then we get (2')

$$\begin{aligned}(|\sigma_2| -_s \Upsilon\sigma_1) -_s \Upsilon\beta &= |\sigma_2| -_s (\Upsilon\sigma_1 \cup \Upsilon\beta) && \text{Lemma 2(2)}\\ &= \{\#\alpha\#\} -_s (\Upsilon\sigma_1 \cup \Upsilon\beta) && \sigma_2 = [\alpha] \text{ with Definition 7}\\ &\preccurlyeq_{\mathsf{mul}} \{\#\alpha\#\} && \text{Lemma 6(8)}\end{aligned}$$

and (because $\Upsilon\sigma_2 = \Upsilon\alpha$), similar as in the other case from (3) we get

$$((|\sigma_3| -_s \Upsilon\sigma_2) -_s \Upsilon\sigma_1) -_s \Upsilon\beta = \{\#\} \tag{3"}$$

From (1), (2'), and (3") we conclude ($\star$). ◀

Next we prepare for the key lemma to establish (ii), i.e., Lemma 17, after establishing useful intermediate results. Note that Lemma 16(2) can be seen as an inverse of Lemma 14.

▶ **Lemma 16.**
1. $\alpha \in\# |\sigma| \Rightarrow \exists\sigma_1\sigma_3.\ \sigma = \sigma_1\alpha\sigma_3 \wedge \alpha \notin \Upsilon\sigma_1$
2. $|\sigma| \subseteq \Upsilon S \Rightarrow \sigma \subseteq \Upsilon S$
3. $S \subseteq \Upsilon T \Rightarrow \Upsilon S \subseteq \Upsilon T$

Proof.
1. By induction on σ. The base case is trivial. In the step case we can assume that $\alpha\in\#|\beta\sigma|$. We proceed by case analysis.
   - If $\alpha = \beta$ then we are done with $\sigma_1 = [\,]$ and $\sigma_3 = \sigma$.
   - In the other case we have $\alpha \in\# |\sigma|$ and $\alpha \notin \Upsilon\beta$ from Definition 7. The induction hypothesis yields σ_1' and σ_3' with $\sigma = \sigma_1'\alpha\sigma_3'$ such that $\alpha \notin \Upsilon\sigma_1'$. Because $\alpha \notin \Upsilon\beta$ we can conclude with $\sigma_1 = \beta\sigma_1'$ and $\sigma_3 = \sigma_3'$ using Lemma 6(1).
2. Assume $\alpha \in \sigma$. If $\alpha \in\# |\sigma|$ then we are done by the hypothesis. In the other case there must be a $\beta \in |\sigma|$ (easy induction on σ) with $\alpha \prec \beta$. From the hypothesis we get that $\beta \in \Upsilon S$ and by transitivity also $\alpha \in \Upsilon S$, which finishes the proof.
3. By monotonicity of Υ ([14, Proposition 1.4.8(2)]) the assumption yields $\Upsilon S \subseteq \Upsilon(\Upsilon T)$. Lemma 4 finishes the proof. ◀

With Lemma 16 we can now prove the following key result to establish (ii):

▶ **Lemma 17.** $|\sigma'| -_s \Upsilon\beta \preccurlyeq_{mul} \{\#\alpha\#\} \Rightarrow \exists\sigma_1\sigma_2\sigma_3.\ \sigma' = \sigma_1\sigma_2\sigma_3 \wedge \sigma_1 \subseteq \Upsilon\beta \wedge \mathit{length}\ \sigma_2 \leq 1 \wedge \sigma_2 \subseteq \{\alpha\} \wedge \sigma_3 \subseteq \Upsilon\alpha\beta$

Proof. To show the result we perform a case analysis.
- If $\alpha \in\# |\sigma'| -_s \Upsilon\beta$ then Lemma 16(1) yields σ_1 and σ_3 with $\sigma' = \sigma_1\alpha\sigma_3$ and $\alpha \notin \Upsilon\sigma_1$. Hence from the hypothesis and Lemma 8(2) we get

$$(|\sigma_1| -_s \Upsilon\beta) + \{\#\alpha\#\} + (((|\sigma_3| -_s \Upsilon\alpha) -_s \Upsilon\sigma_1) -_s \Upsilon\beta) \preccurlyeq_{\mathsf{mul}} \{\#\alpha\#\}$$

and since $\alpha \notin \Upsilon\sigma_1$ and $\alpha \notin \Upsilon\beta$ it follows that

$$|\sigma_1| -_s \Upsilon\beta = \{\#\} \text{ and } ((|\sigma_3| -_s \Upsilon\alpha) -_s \Upsilon\sigma_1) -_s \Upsilon\beta = \{\#\}$$

Now, Lemma 2(2) yields

$$|\sigma_1| \subseteq \Upsilon\beta \text{ and } |\sigma_3| \subseteq \Upsilon\alpha \cup \Upsilon\sigma_1 \cup \Upsilon\beta$$

and from Lemma 16(2) we get

$$\sigma_1 \subseteq \Upsilon\beta \text{ and } \sigma_3 \subseteq \Upsilon\alpha \cup \Upsilon\sigma_1 \cup \Upsilon\beta$$

The latter simplifies to $\sigma_3 \subseteq \Upsilon\alpha\beta$ using $\Upsilon\sigma_1 \subseteq \Upsilon\beta$ (from Lemma 16(3)) and Lemma 6(1). Hence in this case the result follows with $\sigma_2 = [\alpha]$.

- If $\alpha \notin\# |\sigma'| -s \Upsilon\beta$

$$\begin{aligned} &\Rightarrow |\sigma'| -s \Upsilon\beta \subseteq \Upsilon\alpha && \text{hypothesis} \\ &\Rightarrow |\sigma'| \subseteq \Upsilon\alpha\beta && \text{Lemma 6(1)} \\ &\Rightarrow \sigma' \subseteq \Upsilon\alpha\beta && \text{Lemma 16(2)} \end{aligned}$$

In this case the result follows with empty σ_1, empty σ_2, and $\sigma' = \sigma_3$. ◀

Now Lemma 13 follows from Lemma 15 (LD′ ⇒ LD) and Lemma 17 (LD ⇒ LD′).

3.4 Labeled Rewriting

So far we have only considered sequences of labels. However, for the main result (Section 3.5) we need labeled rewriting. Hence this section sketches how we formalized *labeled* (abstract) rewriting before lifting the results from Section 3.2 from labels to labeled rewriting (a step which is left implicit in [13]). In the theory Abstract_Rewriting.thy an *abstract rewrite system* (ARS) is a set of pairs of objects of the same type, i.e., a binary relation. Confluence is also defined in Abstract_Rewriting.thy, but the theory does not provide support for *labeled* abstract rewrite systems. In the sequel we write $\mathcal{A}$ ($\mathcal{B}$) for (labeled) ARSs. A *labeled ARS* $\mathcal{B}$ is a ternary relation. We call $(a, \alpha, b) \in \mathcal{B}$ a *(labeled rewrite) step* and write $a \xrightarrow{\alpha} b$. Next we define *(labeled rewrite) sequences* inductively, i.e., for each object a there is the empty sequence $a \xrightarrow{[]}\!\!\!\!\rightarrow a$ and if $a \xrightarrow{\alpha} b$ is a step and $b \xrightarrow{\sigma}\!\!\!\!\rightarrow c$ is a sequence then $a \xrightarrow{\alpha\sigma}\!\!\!\!\rightarrow c$ is a sequence.

▶ **Example 18.** Let $\mathcal{B}$ be the labeled ARS $\{(a, \alpha, b), (b, \beta, c)\}$. Then $a \xrightarrow{\alpha} b \xrightarrow{\beta} c$ (or $a \xrightarrow{\alpha\beta}\!\!\!\!\rightarrow c$) is a sequence in $\mathcal{B}$. The empty sequence $a \xrightarrow{[]}\!\!\!\!\rightarrow a$ we also write as a.

We prove useful properties for sequences, i.e., that chopping off a segment of a sequence again yields a sequence and that two sequences can be concatenated (provided the last element of the first sequence coincides with the first element of the second sequence).

▶ **Lemma 19.** *Let* $a_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} a_n$ *and* $b_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} b_m$ *be sequences.*

1. *Then* $a_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{i-1}} a_i$ *and* $a_i \xrightarrow{\alpha_i} \cdots \xrightarrow{\alpha_{n-1}} a_n$ *are sequences for any* $1 \leqslant i \leqslant n$.
2. *If* $a_n = b_1$ *then* $a_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} a_n = b_1 \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{m-1}} b_m$ *is a sequence.*

Proof. By induction on $a_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} a_n$. ◀

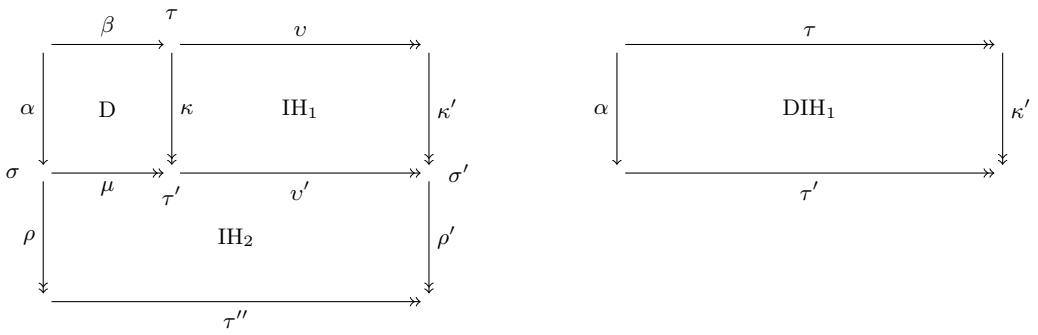
As a next step we introduce diagrams.

▶ **Definition 20.** A *diagram* is a quadruple of sequences $(\xrightarrow{\tau}\!\!\!\!\rightarrow, \xrightarrow{\sigma}\!\!\!\!\rightarrow, \xrightarrow{\sigma'}\!\!\!\!\rightarrow, \xrightarrow{\tau'}\!\!\!\!\rightarrow)$ such that the start and endpoints of the sequences satisfy the picture in Figure 1a. A diagram is called *decreasing* if its labels are.

We lift Lemma 11 from labels to diagrams.

▶ **Lemma 21** ([13, Lemma 3.5] for decreasing diagrams)**.** *Pasting two decreasing diagrams yields a decreasing diagram. For a picture see Figure 2a.*

Proof. With the help of Lemma 19(2) we show that pasting two diagrams again yields a diagram. That pasting preserves decreasingness follows from Lemma 11. ◀



(a) Local decreasingness implies decreasingness. **(b)** Pasting D and IH_1 into DIH_1.

Figure 4 Lemma 26

3.5 Main Result

We establish that if all local peaks of a labeled ARS $\mathcal{B}$ are decreasing then all peaks of $\mathcal{B}$ are decreasing, following the structure of the proof of [13, Theorem 3.7]. (Changes are discussed in Section 5). Note that only here we need that $\prec$ is well-founded, from which irreflexivity immediately follows (to satisfy our global assumption from Section 2). First we introduce (local) peaks.

▶ **Definition 22.** A *peak* $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})$ is a pair of labeled rewrite sequences which originate from the same object. A *local peak* is a peak where the sequences consist of a single step.

To prove the main result we introduce a measure on *peaks* (actually on pairs of sequences).

▶ **Definition 23.** Let $|(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})| := |\tau| + |\sigma|$. Then we can lift $\prec$ as a relation on labels to a relation on pairs of sequences $\prec_{\mathsf{peak}}$, i.e., $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow}) \prec_{\mathsf{peak}} (\overset{\tau'}{\twoheadrightarrow}, \overset{\sigma'}{\twoheadrightarrow})$ if $|(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})| \prec_{\mathsf{mul}} |(\overset{\tau'}{\twoheadrightarrow}, \overset{\sigma'}{\twoheadrightarrow})|$.

For proofs of induction we establish that $\prec_{\mathsf{peak}}$ is well-founded.

▶ **Lemma 24.** *Let $\prec$ be well-founded. Then $\prec_{peak}$ is well-founded.*

Proof. From [4] we get that $\prec_{\mathsf{mul}}$ is well-founded (this proof is contained in Multiset.thy). We proceed by contraposition. Assume the measure on peaks is not well-founded. Then we obtain an infinite sequence $\cdots \prec_{\mathsf{peak}} (\tau_2, \sigma_2) \prec_{\mathsf{peak}} (\tau_1, \sigma_1)$ which entails an infinite sequence on multisets $\cdots \prec_{\mathsf{mul}} |\tau_2| + |\sigma_2| \prec_{\mathsf{mul}} |\tau_1| + |\sigma_1|$ showing the result. ◀

▶ **Definition 25.** A peak $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})$ in a labeled ARS is *decreasing* if it can be completed into a decreasing diagram, i.e., there are $\overset{\sigma'}{\twoheadrightarrow}$ and $\overset{\tau'}{\twoheadrightarrow}$ such that the conditions of Figure 1a are satisfied. A peak is *locally decreasing*, if it is decreasing and a local peak.

▶ **Lemma 26** (similar to [13, Theorem 3.7])**.** *Let $\mathcal{B}$ be a labeled ARS and $\prec$ be a transitive and well-founded relation on the labels. If all local peaks of $\mathcal{B}$ are decreasing, then all peaks of $\mathcal{B}$ are decreasing.*

Proof. To show that all peaks are decreasing we fix a peak $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})$ and show that this peak can be completed into a decreasing diagram. The proof is by well-founded induction on $\prec_{\mathsf{peak}}$ and there only is the step case. The interesting situation is when neither τ nor σ are empty, i.e., (using Lemma 19(1) we obtain) $\overset{\tau}{\twoheadrightarrow} = \overset{\beta}{\to} \cdot \overset{\upsilon}{\twoheadrightarrow}$ and $\overset{\sigma}{\twoheadrightarrow} = \overset{\alpha}{\to} \cdot \overset{\rho}{\twoheadrightarrow}$ (see Figure 4a). Hence $(\overset{\beta}{\to}, \overset{\alpha}{\to})$ is a local peak and from the assumption we obtain a decreasing diagram with joining

sequences $\overset{\kappa}{\twoheadrightarrow}$ and $\overset{\mu}{\twoheadrightarrow}$. We obtain that $(\overset{\upsilon}{\twoheadrightarrow}, \overset{\kappa}{\twoheadrightarrow})$ is a peak and want to show that the measure of this peak is smaller than that of $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})$ (to apply the induction hypothesis). Since β is not empty with Lemma 12 we establish that $|(\overset{\upsilon}{\twoheadrightarrow}, \overset{\kappa}{\twoheadrightarrow})|$ is smaller than $|(\overset{\tau}{\twoheadrightarrow}, \overset{\alpha}{\to})|$ and from $|\alpha| \preccurlyeq_{\mathsf{mul}} |\sigma|$[3] we obtain the desired result. Now, the induction hypothesis yields that IH_1 is a decreasing diagram. Concatenating (using Lemma 19(2)) $\overset{\mu}{\twoheadrightarrow}$ and $\overset{\upsilon'}{\twoheadrightarrow}$ into a sequence $\overset{\tau'}{\twoheadrightarrow}$, using Lemma 21 we can paste the diagrams D and IH_1 into a decreasing diagram (DIH_1, see Figure 4b). The peak $(\overset{\tau'}{\twoheadrightarrow}, \overset{\rho}{\twoheadrightarrow})$ is smaller than the peak $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})$ by a mirrored version of Lemma 12 and hence the induction hypothesis yields the decreasing diagram IH_2. Finally, a mirrored version of Lemma 21 pastes DIH_1 and IH_2 into a decreasing diagram. ◀

We define local decreasingness for ARSs.

▶ **Definition 27** ([13, Definition 3.8])**.** An ARS $\mathcal{A}$ is *locally decreasing* if there exists a transitive and well-founded relation $\prec$ on the labels such that all local peaks are decreasing for (a labeled version of) $\mathcal{A}$.

Finally we arrive at the main result for soundness:

▶ **Corollary 28** ([13, Corollary 3.9])**.** *A locally decreasing ARS is confluent.*

Proof. From local decreasingness we get a transitive and well-founded relation $\prec$ such that all local peaks are decreasing in a labeled version of the ARS. Lemma 26 yields that all peaks are decreasing. The result follows by dropping labels from the labeled rewrite sequences. ◀

4 Formalization of the Conversion Version

In this section we give a formal proof for the main result underlying that local decreasingness with respect to conversions (see [15]) implies confluence. To this end we formally introduce (labeled) conversions, similarly to labeled rewrite sequences. For each object a there is the empty conversion $a \overset{\Box}{\leftrightarrow\!\!\!\!\twoheadrightarrow} a$ (also just written a) and if $a \overset{\alpha}{\to} b$ ($a \overset{\alpha}{\leftarrow} b$) is a labeled rewrite step and $b \overset{\sigma}{\leftrightarrow\!\!\!\!\twoheadrightarrow} c$ is a conversion then $a \overset{\alpha}{\to} b \overset{\sigma}{\leftrightarrow\!\!\!\!\twoheadrightarrow} c$ ($a \overset{\alpha}{\leftarrow} b \overset{\sigma}{\leftrightarrow\!\!\!\!\twoheadrightarrow} c$) is a conversion (often written $a \overset{\alpha\sigma}{\leftrightarrow\!\!\!\!\twoheadrightarrow} c$). For conversions we prove similar properties as for sequences (see Lemma 19). In addition we establish that mirroring a conversion again yields a conversion (with the same set of labels) and that every sequence is a conversion.

▶ **Lemma 29.** *Let* $a_1 \overset{\alpha_1}{\leftrightarrow} \cdots \overset{\alpha_{n-1}}{\leftrightarrow} a_n$ *and* $b_1 \overset{\beta_1}{\leftrightarrow} \cdots \overset{\beta_{m-1}}{\leftrightarrow} b_m$ *be conversions.*

1. *Then* $a_1 \overset{\alpha_1}{\leftrightarrow} \cdots \overset{\alpha_{i-1}}{\leftrightarrow} a_i$ *and* $a_i \overset{\alpha_i}{\leftrightarrow} \cdots \overset{\alpha_{n-1}}{\leftrightarrow} a_n$ *are conversions for any* $1 \leqslant i \leqslant n$.
2. *If* $a_n = b_1$ *then* $a_1 \overset{\alpha_1}{\leftrightarrow} \cdots \overset{\alpha_{n-1}}{\leftrightarrow} a_n = b_1 \overset{\beta_1}{\leftrightarrow} \cdots \overset{\beta_{m-1}}{\leftrightarrow} b_m$ *is a conversion.*
3. *Then* $a_n \overset{\alpha_{n-1}}{\leftrightarrow} \cdots \overset{\alpha_1}{\leftrightarrow} a_1$ *is a conversion and* $\{\alpha_1, \ldots, \alpha_n\} = \{\alpha_n, \ldots, \alpha_1\}$.
4. *If* $c_1 \overset{\gamma_1}{\to} \cdots \overset{\gamma_{n-1}}{\to} c_n$ *is a sequence then* $c_1 \overset{\gamma_1}{\leftrightarrow} \cdots \overset{\gamma_{n-1}}{\leftrightarrow} c_n$ *is a conversion.*

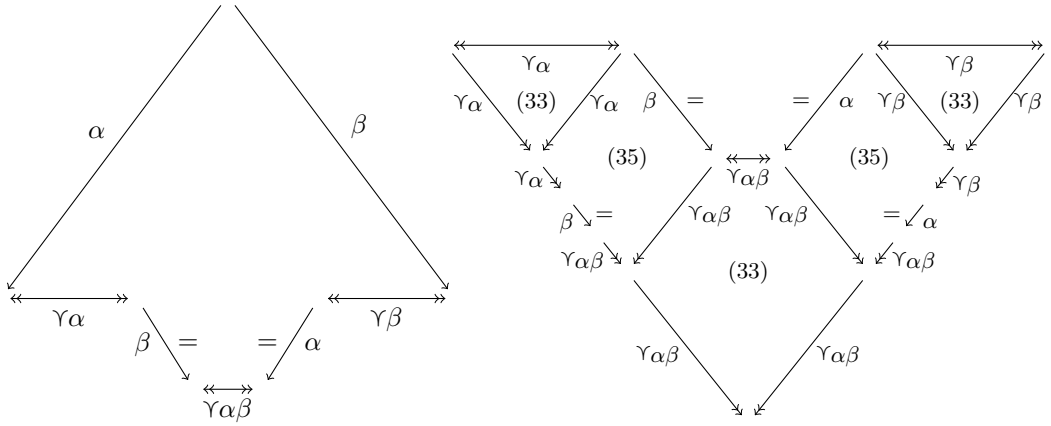
Proof. Items (1)-(3) are proved by induction on the first conversion, item (4) is proved by induction on the sequence. ◀

We will also use the following easy lemma being a direct consequence of Definition 3.

▶ **Lemma 30.** *If* $M \preccurlyeq_{mul} N$ *and* $N \subseteq \Upsilon S$ *then* $M \subseteq \Upsilon S$. ◀

The following result (stated as observation in [15]) follows from Lemma 30.

[3] This step is not mentioned in [13, 14] but hinted at in [15].



(a) Local decreasingness wrt. conversions. **(b)** Closing the conversion into a valley.

Figure 5 Conversion version of decreasing diagrams.

▶ **Lemma 31.** *If* $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow}, \overset{\sigma'}{\twoheadrightarrow}, \overset{\tau'}{\twoheadrightarrow})$ *is a decreasing diagram and* $|(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow})| \subseteq \Upsilon M$ *then also* $|(\overset{\sigma'}{\twoheadrightarrow}, \overset{\tau'}{\twoheadrightarrow})| \subseteq \Upsilon M$. ◀

A local peak $(\overset{\beta}{\to}, \overset{\alpha}{\to})$ is *decreasing with respect to conversions*[4] if there exist conversions such that the constraints from Figure 5a are satisfied. Now we can state the main result underlying soundness of the conversion version of decreasing diagrams.

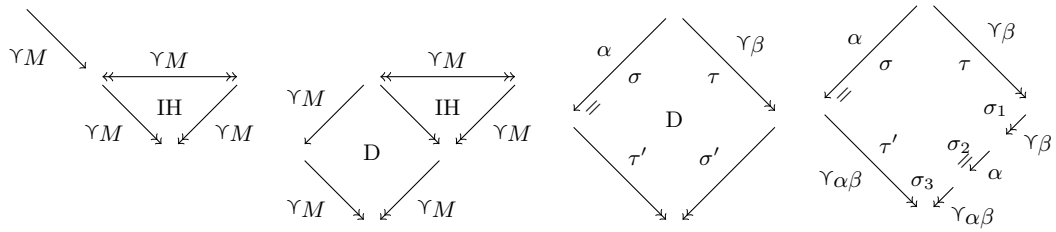
▶ **Lemma 32.** *Let* $\mathcal{B}$ *be a labeled ARS and* $\prec$ *be a transitive and well-founded relation on the labels. If all local peaks of* $\mathcal{B}$ *are decreasing with respect to conversions, then all peaks of* $\mathcal{B}$ *are decreasing (with respect to valleys).*

Proof. Similar to [15] we follow the proof of the valley version (see Lemma 26). In contrast to Lemma 26 we do not get decreasingness of the local peak $(\overset{\beta}{\to}, \overset{\alpha}{\to})$ (in Figure 4a) by assumption. Instead our assumption yields local decreasingness with respect to conversions, i.e., as depicted in Figure 5a. We close the conversion into a valley as outlined in Figure 5b. To this end we use Lemmata 33 and 35 (see below) and conclude the valleys as shown in Figure 5b. Note that for the final application of Lemma 33 we apply Lemma 29 first, to combine the sequences and conversions into a single conversion. Lemma 13 (lifted to rewriting sequences) then shows decreasingness of the diagram. ◀

The main structure of our proof follows the one from [15]. However, there the proofs of two key results are sketchy and informal. We identified the statements as Lemmata 33 and 35 and provide formal proofs. Note that to establish these properties we can use the induction hypothesis (from the proof of Lemma 32), e.g., peaks whose measure is smaller than $|(\overset{\beta}{\to}, \overset{\alpha}{\to})|$ can be completed into a decreasing diagram.

▶ **Lemma 33.** *Let all peaks smaller than* $|(\overset{\beta}{\to}, \overset{\alpha}{\to})|$ *have a decreasing diagram. Then for any* M *with* $M \preccurlyeq_{\mathsf{mul}} \{\#\alpha, \beta\#\}$ *we have* $\overset{\Upsilon M}{\twoheadleftarrow\!\!\twoheadrightarrow} \subseteq \overset{\Upsilon M}{\twoheadrightarrow} \cdot \overset{\Upsilon M}{\twoheadleftarrow}$.

[4] Please note the asymmetry to the definition of local decreasingness (Definition 25).



(a) Lemma 33 (case $\to$). **(b)** Lemma 33 (case $\leftarrow$). **(c)** Lemma 35. **(d)** Lemma 35.

Figure 6 Lemmata 33 and 35.

Proof. By induction on the conversion $\overset{\Upsilon M}{\twoheadleftarrow\!\!\twoheadrightarrow}$. The base case is trivial. In the step case we have $\overset{\Upsilon M}{\leftrightarrow} \cdot \overset{\Upsilon M}{\twoheadleftarrow\!\!\twoheadrightarrow}$. The induction hypothesis yields $\overset{\Upsilon M}{\leftrightarrow} \cdot \overset{\Upsilon M}{\twoheadrightarrow} \cdot \overset{\Upsilon M}{\twoheadleftarrow}$. We consider two cases. If the first step is from left to right, i.e., $\overset{\Upsilon M}{\to}$ then the result follows from Lemma 29(2) (see Figure 6a). In the other case we have $\overset{\Upsilon M}{\leftarrow} \cdot \overset{\Upsilon M}{\twoheadrightarrow} \cdot \overset{\Upsilon M}{\twoheadleftarrow}$. Since the peak $\overset{\Upsilon M}{\leftarrow} \cdot \overset{\Upsilon M}{\twoheadrightarrow}$ has a smaller measure than $(\overset{\beta}{\to}, \overset{\alpha}{\to})$ it can be completed into a decreasing diagram and Lemma 31 in combination with Lemma 29(2) yields the result (see Figure 6b). ◀

To show the second key result we establish a useful decomposition result on sequences.

▶ **Lemma 34.** *Let $\overset{\sigma}{\twoheadrightarrow}$ be a sequence and $\sigma = \sigma_1\sigma_2$. Then there are sequences $\overset{\sigma_1}{\twoheadrightarrow}$ and $\overset{\sigma_2}{\twoheadrightarrow}$ such that $\overset{\sigma}{\twoheadrightarrow} = \overset{\sigma_1}{\twoheadrightarrow} \cdot \overset{\sigma_2}{\twoheadrightarrow}$.*

Proof. By induction on the sequence $\overset{\sigma}{\twoheadrightarrow}$. ◀

Below $\overset{\alpha}{\to}{}^{=}$ stands for $\overset{\alpha}{\to}$ (one step) or $\overset{[]}{\twoheadrightarrow}$ (zero steps). Please note the similarity of the following result to the explicit characterization of local decreasingness (cf. Figure 3a).

▶ **Lemma 35.** *Let all peaks smaller than $|(\overset{\beta}{\to}, \overset{\alpha}{\to})|$ have a decreasing diagram. Then the peak $(\overset{\Upsilon\beta}{\twoheadrightarrow}, \overset{\alpha}{\to}{}^{=})$ can be closed by $\overset{\Upsilon\alpha\beta}{\twoheadrightarrow} \cdot \overset{\Upsilon\alpha\beta}{\twoheadleftarrow} \cdot {}^{=}\overset{\alpha}{\leftarrow} \cdot \overset{\Upsilon\beta}{\twoheadleftarrow}$ (see Figure 6d).*

Proof. Since $|(\overset{\Upsilon\beta}{\twoheadrightarrow}, \overset{\alpha}{\to}{}^{=})|$ is smaller than $|(\overset{\beta}{\to}, \overset{\alpha}{\to})|$, it can be completed into a decreasing diagram $(\overset{\tau}{\twoheadrightarrow}, \overset{\sigma}{\twoheadrightarrow}, \overset{\sigma'}{\twoheadrightarrow}, \overset{\tau'}{\twoheadrightarrow})$ (see Figure 6c). First we show $\tau' \subseteq \Upsilon\alpha\beta$. From decreasingness and Lemma 10 we get $|\tau'| \mathrel{-s} \Upsilon\sigma \preccurlyeq_{\mathsf{mul}} |\tau|$. The assumption $\tau \subseteq \Upsilon\beta$ and Lemma 14 yields $|\tau| \subseteq \Upsilon\beta$. Using Lemma 30 we obtain $|\tau'| \mathrel{-s} \Upsilon\sigma \subseteq \Upsilon\beta$, i.e. $|\tau'| \subseteq \Upsilon\beta \cup \Upsilon\sigma$. The assumption $\sigma \subseteq \alpha$ yields $\Upsilon\sigma \subseteq \Upsilon\alpha$ and hence we conclude by Lemmata 6(1) and 16(2).

Next we show that $\overset{\sigma'}{\twoheadrightarrow}$ can be decomposed into $\overset{\sigma_1}{\twoheadrightarrow}$, $\overset{\sigma_2}{\to}{}^{=}$, and $\overset{\sigma_3}{\twoheadrightarrow}$ with $\sigma_1 \subseteq \Upsilon\beta$, $\sigma_2 \subseteq \{\alpha\}$, $\mathsf{length}\ \sigma_2 \leq 1$, and $\sigma_3 \subseteq \Upsilon\alpha\beta$. To this end we first observe that Lemma 17 also holds if β is not a single label but a sequence (here τ). Then from decreasingness we obtain $\sigma' = \sigma_1\sigma_2\sigma_3 \wedge \sigma_1 \subseteq \Upsilon\tau \wedge \mathsf{length}\ \sigma_2 \leq 1 \wedge \sigma_2 \subseteq \{\alpha\} \wedge \sigma_3 \subseteq \Upsilon\alpha\sigma$. Lemma 34 lifts the decomposition of labels to a decomposition of sequences and we can conclude. ◀

An ARS $\mathcal{A}$ is *locally decreasing with respect to conversions* if there exists a transitive and well-founded relation $\prec$ on the labels such that all local peaks are decreasing with respect to conversions for (a labeled version of) $\mathcal{A}$. Finally we arrive at the main result for soundness:

▶ **Corollary 36** ([15, Theorem 3])**.** *A locally decreasing with respect to conversions ARS is confluent.* ◀

5 Meanderings

In this section we discuss differences between our formalization and (proofs from) [13, 15].

Within Isabelle (Abstract_Rewriting.thy) an ARS is a binary relation while in [13] the ARS also contains the domain of the relation. A similar statement holds for labeled ARSs.

General multisets are used in [13], which can represent sets and finite multisets in one go wheres our formalization clearly separates the two concepts. The reason is purely practical, i.e., the Isabelle library already contains the dedicated theories Set.thy and Multiset.thy. The only (negligible) disadvantage we have experienced from this design choice is the need for multiple definitions of the down-set (for lists, sets, and multisets) and for Lemma 6(1). On the other hand, this saved us from formalizing *general multisets*, which we anticipate as a significant endeavor on its own. Moreover, [13] uses a different multiset extension than Multiset.thy. The latter defines the multiset extension as the transitive closure of the "one-step" multiset extension.

▶ **Definition 37.** The *one-step multiset extension* (denoted by $\prec_{\mathsf{mult1}}$) of $\prec$ is defined by

$$M \prec_{\mathsf{mult1}} N \text{ if } \exists\, a\ I\ K.\ M = I + K,\ N = I + \{\#a\#\},\ \forall\, b \in K.\ b \prec a$$

and the *multiset extension* of $\prec$ (denoted by $\prec_{\mathsf{mult}}$) is the transitive closure of $\prec_{\mathsf{mult1}}$.

Based on the results in Multiset.thy and Definition 3(1) we have proven these two definitions equivalent for any transitive base relation.

▶ **Lemma 38.** *If $\prec$ is transitive then $\prec_{mult}$ and $\prec_{mul}$ coincide.* ◀

Moreover we proved the claim in Definition 3.

▶ **Lemma 39.** *We have that $\preccurlyeq_{mul}$ is the reflexive closure of $\prec_{mul}$.* ◀

Proof. First we show the inclusion from left to right. Let $M \preccurlyeq_{\mathsf{mul}} N$. If $J = \{\#\}$ then $M = N$ and the result follows. If $J \neq \{\#\}$ then $M \prec_{\mathsf{mul}} N$ and we are done.

For the reverse inclusion let (M, N) be in the reflexive closure of $\prec_{\mathsf{mul}}$. If $M = N$ then we finish with $I = M$, $K = J = \{\#\}$. In the other case we get suitable I, J, and K from the definition of $\prec_{\mathsf{mul}}$. ◀

Our formalization is first performed for sequences (of labels) and then lifted to labeled rewrite sequences (conversions), a step which is left implicit in [13]. After introducing labeled rewriting, we proved useful results in Isabelle (Lemmata 19 and 29).

In addition to the algebraic proof of Lemma 6(3) from [13] our formalization contains an alternative one. Our proof of Lemma 8(1) differs from the informal one in [13]. Also the formal proof of Lemma 13 differs from the sketch given for [13, Proposition 3.4], requiring auxiliary results (Lemmata 14 and 16).

There are some (tiny) differences between [13, Theorem 3.7] and Lemma 26. In [13] a measure on *diagrams* is used. However, since the closing/joining steps of the diagram are just obtained by the induction hypothesis the measure must be on *peaks* (which is used in [15]). Moreover, since in either case the measure is a multiset it is hard to relate arbitrary multisets to a peak. Hence we lifted the order on labels $\prec$ to peaks $\prec_{\mathsf{peak}}$ (Section 3.5) and used well-founded induction on this order. In the formalization of Lemma 26 (Footnote 3) we identified a necessary step to apply the induction hypothesis. Another aspect where our formalization deviates from [13] is that the original work uses families of labeled ARSs whereas our formalization considers a single labeled ARS only. Hence [13, Theorem 3.7]

states the main result on families of ARSs whereas our Lemma 26 makes a statement about a single ARS.

Concerning [13] our formal proofs for the alternative formulation of local decreasingness (Lemma 13) differs from the one in [13, 14]. While this alternative formulation of local decreasingness was not needed to obtain the main result underlying the valley version ([13, Main Theorem 3.7], i.e., Lemma 26), it was (in a generalized formulation) essential for the main result underlying the conversion version ([15, Theorem 3], i.e., Lemma 32). Furthermore we gave formal proofs for two (informal) key observations made in the proof of [15, Theorem 3], resulting in Lemmata 33 and 35. Especially the latter has a non-trivial formal proof, since the induction hypothesis yields decreasingness (see Figure 6c) but not the desired decomposition of the joining sequences (see Figure 6d), in contrast to what the proof in [15] conveys.

6 Conclusion

In this paper we have described a formalization of decreasing diagrams in the theorem prover Isabelle following the original proofs from [13, 15]. In Sections 3.3 and 3.4 our formal proofs deviate from the either informal or implicit ones in [13] and we also elaborate on Lemma 35, a result which is implicitly used in [15]. To show the applicability of our formalization we performed a mechanical proof of Newman's lemma using decreasing diagrams (following [13, Corollary 4.4]). Our formalization has few dependencies on existing theories. From Abstract_Rewriting.thy we employ some properties for unlabeled abstract rewriting (and the definition of confluence). The theory Multiset.thy provides standard multiset operations and a well-foundedness proof of the multiset extension of a well-founded relation. Note that some of our results on multisets (a formalized proof of [13, Lemma 2.6(3)], i.e., Lemma 6(3)) might be of interest for a larger community.

In [2] a "point version" of decreasing diagrams is introduced, where objects are labeled instead of steps. It is unknown if the point version is equivalent to the standard one. Parts of [2] have been formalized in Coq but 29 axioms are assumed, i.e., not proven in the theorem prover. Furthermore the more useful alternative representation of local decreasingness (Lemma 13) is not considered in [2]. The same holds for the conversion version. Hence [2] is only a *partial* formalization and essentially different from ours.

We anticipate that our contribution paves the way for future work in several directions. One possibility is the formalization of confluence results that can be proven with decreasing diagrams (e.g. Toyama's theorem [26]). The benefit might be two-fold. On the one hand side the proof by decreasing diagrams might be easier to formalize and furthermore proofs by decreasing diagrams are constructive, cf. [16]. Another idea would be the certification of confluence proofs (based on decreasing diagrams) given by automated confluence provers.[5] Both aims require to lift our formalization from abstract rewriting to term rewriting, which is a natural idea for future work.

Acknowledgments

This research is supported by FWF P22467. We thank the anonymous reviewers, Bertram Felgenhauer, Nao Hirokawa, and Aart Middeldorp for helpful comments. Bertram Felgenhauer contributed an initial proof of Lemma 6(3) and located the formalization of [2].

[5] Certification is already established in the termination community where it has shown tools as well as termination criteria unsound.

References

1 Bezem, M., Klop, J., V. van Oostrom: Diagram techniques for confluence. I&C 141(2), 172–204 (1998)
2 Bognar, M.: A point version of decreasing diagrams. In: Proceedings Accolade 1996. Dutch Graduate School in Logic. pp. 1–14 (1997). The formalization is available from `http://web.archive.org/web/20051226052550/http://www.cs.vu.nl/~mirna/`
3 Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Automated certified proofs with CiME3. In: Proc. 22nd RTA. LIPIcs, vol. 10, pp. 21–30 (2011)
4 Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Comm. ACM 22(8), 465–476 (1979)
5 Felgenhauer, B.: A proof order for decreasing diagrams. In: Proc. 1st IWC. pp. 7–14 (2012)
6 Galdino, A., Ayala-Rincón, M.: A formalization of Newman's and Yokouchi's lemmas in a higher-order language. JFR 1(1), 39–50 (2008)
7 Galdino, A., Ayala-Rincón, M.: A formalization of the Knuth-Bendix(-Huet) critical pair theorem. JAR 45(3), 301–325 (2010)
8 Huet, G.: Residual theory in lambda-calculus: A formal development. JFP 4(3), 371–394 (1994)
9 Jouannaud, J.P., van Oostrom, V.: Diagrammatic confluence and completion. In: Proc. 36th ICALP. LNCS, vol. 5556, pp. 212–222 (2009)
10 Klop, J., van Oostrom, V., de Vrijer, R.: A geometric proof of confluence by decreasing diagrams. JLP 10(3), 437–460 (2000)
11 Nipkow, T.: More Church-Rosser proofs. JAR 26(1), 51–66 (2001)
12 Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. vol. 2283 of LNCS. Springer (2002)
13 van Oostrom, V.: Confluence by decreasing diagrams. TCS 126(2), 259–280 (1994)
14 van Oostrom, V.: Confluence for Abstract and Higher-Order Rewriting. PhD thesis, Vrije Universiteit, Amsterdam (1994)
15 van Oostrom, V.: Confluence by decreasing diagrams – converted. In: Proc. 19th RTA. LNCS, vol. 5117, pp. 306–320 (2008)
16 van Oostrom, V.: Modularity of confluence constructed. In: Proc. 4th IJCAR. LNCS, vol. 5195, pp. 348–363 (2008)
17 van Oostrom, V.: Decreasing proof orders – interpreting conversions in involutive monoids. In: Proc. 1st IWC. pp. 1–4 (2012)
18 Pfenning, F.: A proof of the Church-Rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, School of Computer Science, Carnegie Mellon University (1992)
19 Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., Martín-Mateos, F.J.: Formal proofs about rewriting using ACL2. AMAI 36(3), 239–262 (2002)
20 Shankar, N.: A mechanical proof of the Church-Rosser theorem. JACM 35(3), 475–522 (1988)
21 Sternagel, C., Thiemann, R.: Abstract rewriting. AFP (2010)
22 Støvring, K.: Extending the extensional lambda calculus with surjective pairing is conservative. LMCS 2(2), 14 pages (2006)
23 Takahashi, M.: Parallel reductions in λ-calculus. I&C 118(1), 120–127 (1995)
24 Terese: Term Rewriting Systems. vol. 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003)
25 Thiemann, R.: Certification of confluence proofs using CeTA. In: Proc. 1st IWC. p. 45 (2012)

26 Toyama, Y.: On the Church-Rosser property for the direct sum of term rewriting systems. JACM 34(1), 128–143 (1987)
27 Zankl, H.: Confluence by decreasing diagrams – formalized. CoRR abs/1210.1100v2, 17 pages (2013)