Proceedings of the 21st International Conference on Rewriting Techniques and Applications

RTA'10, July 11–13, 2010, Edinburgh, UK

Edited by

Christopher Lynch

Editor

Christopher Lynch
Department of Math and Computer Science
P.O. Box 5815
Clarkson University
Potsdam, NY 13699-5815, USA
clynch@clarkson.edu

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 978-3-939897-18-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics gGmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

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.

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works license: http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.
- Noncommercial: The work may not be used for commercial purposes.
- No derivation: It is not allowed to alter or transform this work.

The copyright is retained by the corresponding authors.

ISBN 978-3-939897-18-7 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

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.

Editorial Board

- Susanne Albers (Humboldt University Berlin)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Wolfgang Thomas (RWTH Aachen)
- Vinay V. (Chennai Mathematical Institute)
- Pascal Weil (*Chair*, University Bordeaux)
- Reinhard Wilhelm (Saarland University, Schloss Dagstuhl)

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

Preface
Christopher Lynch XIII

Invited Talks

Automata for Data Words And Data Trees
Mikołaj Bojańczyk 1

Realising Optimal Sharing
Vincent Van Oostrom 5

Regular Papers

Automated Confluence Proof by Decreasing Diagrams based on Rule-Labelling
Takahito Aoto 7

Higher-Order (Non-)Modularity
Claus Appel, Vincent van Oostrom, and Jakob Grue Simonsen 17

Closing the Gap Between Runtime Complexity and Polytime Computability
Martin Avanzini and Georg Moser 33

Abstract Models of Transfinite Reductions
Patrick Bahr 49

Partial Order Infinitary Term Rewriting and Böhm Trees
Patrick Bahr 67

Unique Normal Forms in Infinitary Weakly Orthogonal Term Rewriting
Joerg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Jan Willem Klop, and Vincent van Oostrom 85

The Undecidability of Type Related Problems in Type-Free Style System F
Ken-etsu Fujita and Aleksy Schubert 103

On (Un)Soundness of Unravelings
Karl Gmeiner, Bernhard Gramlich, and Felix Schernhammer 119

A Proof Calculus which Reduces Syntactic Bureaucracy
Alessio Guglielmi, Tom Gundersen, and Michel Parigot 135

A Rewriting Logic Semantics Approach to Modular Program Analysis
Mark Hills and Grigore Rosu 151

Infinitary Rewriting: Foundations Revisited
Stefan Kahrs 161

Underspecified Computation of Normal Forms
Alexander Koller and Stefan Thater 177

Order-Sorted Unification with Regular Expression Sorts
Temur Kutsia and Mircea Marin 193

Proceedings of the 21st International Conference on Rewriting Techniques and Applications.
Editor: C. Lynch; pp. v–xii

Leibniz International Proceedings in Informatics
Schloss Dagstuhl Publishing, Germany

An Efficient Nominal Unification Algorithm
Jordi Levy and Mateu Villaret 209

Computing Critical Pairs in 2-Dimensional Rewriting Systems
Samuel Mimram 227

Polynomial Interpretations over the Reals do not Subsume Polynomial Interpretations over the Integers
Friedrich Neurauter and Aart Middeldorp 243

Automated Termination Analysis of Java Bytecode by Term Rewriting
Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl 259

Declarative Debugging of Missing Answers for Maude Specifications
Adrian Riesco, Alberto Verdejo, Narciso Marti-Oliet 277

Simulation in the Call-by-Need Lambda-Calculus with letrec
Manfred Schmidt-Schauss, David Sabel, and Elena Machkasova 295

Weak Convergence and Uniform Normalization in Infinitary Rewriting
Jakob Grue Simonsen 311

Certified Subterm Criterion and Certified Usable Rules
Christian Sternagel and René Thiemann 325

Termination of Linear Bounded Term Rewriting Systems
Marc Sylvestre, Irène Durand, and Géraud Sénizergues 341

Polynomially Bounded Matrix Interpretations
Johannes Waldmann 357

Optimizing $\mathrm{mkb}_{\mathrm{TT}}$
Sarah Winkler, Haruhiko Sato, Aart Middeldorp, and Masahito Kurihara 373

Modular Complexity Analysis via Relative Complexity
Harald Zankl and Martin Korp 385

Proving Productivity in Infinite Data Structures
Hans Zantema and Matthias Raffelsieper 401

Author Index

Aoto, Takahito ... 7
Appel, Claus ... 17
Avanzini, Martin ... 33

Bahr, Patrick ... 49, 67
Bojańczyk, Mikołaj ... 1
Brockschmidt, Marc ... 259

Durand, Irène ... 341

Endrullis, Joerg ... 85

Fujita, Ken-etsu ... 103

Giesl, Jürgen ... 259
Gmeiner, Karl ... 119
Grabmayer, Clemens ... 85
Gramlich, Bernhard ... 119
Guglielmi, Alessio ... 135
Gundersen, Tom ... 135

Hendriks, Dimitri ... 85
Hills, Mark ... 151

Kahrs, Stefan ... 161
Klop, Jan Willem ... 85
Koller, Alexander ... 177
Korp, Martin ... 385
Kurihara, Masahito ... 373
Kutsia, Temur ... 193

Levy, Jordi ... 209

Machkasova, Elena ... 295
Marin, Mircea ... 193
Marti-Oliet, Narciso ... 277
Middeldorp, Aart ... 243, 373
Mimram, Samuel ... 227
Moser, Georg ... 33

Neurauter, Friedrich ... 243

Otto, Carsten ... 259

Parigot, Michel ... 135

Raffelsieper, Matthias ... 401
Riesco, Adrian ... 277
Rosu, Grigore ... 151

Sénizergues, Géraud ... 341
Sabel, David ... 295
Sato, Haruhiko ... 373
Schernhammer, Felix ... 119
Schmidt-Schauss, Manfred ... 295
Schubert, Aleksy ... 103
Simonsen, Jakob Grue ... 17, 311
Sternagel, Christian ... 325
Sylvestre, Marc ... 341

Thater, Stefan ... 177
Thiemann, René ... 325

van Oostrom, Vincent ... 5, 17, 85
Verdejo, Alberto ... 277
Villaret, Mateu ... 209
von Essen, Christian ... 259

Waldmann, Johannes ... 357
Winkler, Sarah ... 373

Zankl, Harald ... 385
Zantema, Hans ... 401

Proceedings of the 21st International Conference on Rewriting Techniques and Applications.
Editor: C. Lynch; pp. vii–xii

Leibniz International Proceedings in Informatics
Schloss Dagstuhl Publishing, Germany

Conference Organization

Conference Chair

Fairouz Kamereddine	Edinburgh, UK

Program Chair

Christopher Lynch	Potsdam, USA

Program Committee

Takahito Aoto	Sendai, Japan
Hubert Comon-lundh	Cachan, France
Dan Dougherty	Worcester, USA
Gilles Dowek	Paris, France
Rachid Echahed	Grenoble, France
Santiago Escobar	Valencia, Spain
Maribel Fernandez	London, UK
Guillem Godoy	Barcelona, Spain
Bernhard Gramlich	Vienna, Austria
Patricia Johann	Glasgow, UK
Fairouz Kamareddine	Edinburgh, UK
Delia Kesner	Paris, France
Georg Moser	Innsbruck, Austria
Hitoshi Ohsaki	Osaka, Japan
Femke van Raamsdonk	Amsterdam, The Netherlands
Sophie Tison	Lille, France
Johannes Waldmann	Leipzig, Germany

RTA Steering Committee

Franz Baader	Dresden, Germany
Hitoshi Ohsaki	Osaka, Japan
Albert Rubio	Barcelona, Spain
Johannes Waldmann	Leipzig, Germany
Ian Mackie	Palaiseau, France
Joachim Niehren	Lille, France

Proceedings of the 21st International Conference on Rewriting Techniques and Applications.
Editor: C. Lynch; pp. ix–xii
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl Publishing, Germany

External Reviewers

Albert, Elvira
Alpuente, María
Ariola, Zena
Avanzini, Martin
Ayala-Rincón, Mauricio
Bahr, Patrick
Balland, Emilie
Berger, Ulrich
Blom, Stefan
Bonelli, Eduardo
Bonfante, Guillaume
Bongaerts, Jochem
Bucciarelli, Antonio
Chiba, Yuki
Contejean, Evelyne
Corradini, Andrea
Creus, Carles
de Vries, Fer-Jan
de Vrijer, Roel
Dershowitz, Nachum
Dyckhoff, Roy
Endrullis, Joerg
Filiot, Emmanuel
Gascón, Adrià
Ghani, Neil
Ghilezan, Silvia
Gmeiner, Karl
Guiraud, Yves
Gutierrez, Raul
Hardin, Therese
Hashimoto, Kenji
Hendriks, Dimitri
Hendrix, Joe
Hym, Samuel
Jay, Barry
Kahrs, Stefan
Korp, Martin
Kuske, Dietrich
Lafont, Yves
Lescanne, Pierre
Lohrey, Markus
Lucas, Salvador
Maneth, Sebastian
Marti-Oliet, Narciso
Middeldorp, Aart
Moura, Flavio
Nakamura, Masaki
Neurauter, Friedrich
Nguyen, Van Tang
Niehren, Joachim
Nishida, Naoki
Palomino, Miguel
Payet, Etienne
Péchoux, Romain
Plump, Detlef
Rahli, Vincent
van Oostrom, Vincent
Rodenburg, Piet
Sakai, Masahiko
Sato, Haruhiko
Schernhammer, Felix
Schnabl, Andreas
Seki, Hiroyuki
Sénizergues, Géraud
Severi, Paula
Simonsen, Jakob Grue
Spoto, Fausto
Sternagel, Christian
Takai, Toshinori
Talbot, Jean-Marc
Thiemann, René
Urbain, Xavier
Urban, Christian
Vidal, German
Yamada, Toshiyuki
Zantema, Hans

Proceedings of the 21st International Conference on Rewriting Techniques and Applications.
Editor: C. Lynch; pp. xi–xii

Leibniz International Proceedings in Informatics
Schloss Dagstuhl Publishing, Germany

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.i

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. XIII-XIV
http://rewriting.loria.fr/rta/

PREFACE

This volume contains the papers presented at the 21st International Conference on Rewriting Techniques and Applications (RTA 2010) which was held from July 11 to July 13, 2010, in Edinburgh, Scotland as part of the 5th International Federated Logic Conference (FLOC 2010), together with the International Joint Conference on Automated Reasoning (IJCAR 2010), the Conference on Automated Verification (CAV 2010), the IEEE Symposium on Logic in Computer Science (LICS 2010), the International Conference on Logic Programming (ICLP 2010), the Conference on Theory and Applications of Satisfiability Testing (SAT 2010), the Conference on Interactive Theorem Proving (ITP 2010) and the Computer Security Foundations Symposium (CSF 2010). Workshops associated with RTA 2010 were the 5th International Workshop on Higher-Order Rewriting (HOR 2010), the Annual Meeting of the IFIP Working Group 1.6 on Term Rewriting, the International Workshop on Strategies in Rewriting, Proving, and Programming (IWS 2010), the 24th International Workshop on Unification (UNIF 2010) and the 11th International Workshop on Termination (WST 2010).

RTA is the major forum for the presentation of research on all aspects of rewriting. Previous RTA conferences were held in Dijon (1985), Bordeaux (1987), Chapel Hill (1989), Como (1991), Montreal (1993), Kaiserslautern (1995), Rutgers (1996), Sitges (1997), Tsukuba (1998), Trento (1999), Norwich (2000), Utrecht (2001), Copenhagen (2002), Valencia (2003), Aachen (2004), Nara (2005), Seattle (2006), Paris (2007), Hagenberg (2008) and Brasilia (2009).

For RTA 2010, 23 regular research papers and three system descriptions were accepted out of 48 submissions. Each paper was reviewed by at least three members of the Program Committee, with the help of 76 external reviewers, and an electronic meeting of the Program Committee was held using Andrei Voronkov's EasyChair system, which was invaluable in the reviewing process, the electronic Program Committee meeting, and the preparation of the conference schedule, and this proceedings.

The Program Committee gave the award for Best Contribution to RTA 2010 to Patrick Bahr for two papers: "Partial Order Infinitary Term Rewriting and Bhm Trees" and " Abstract Models of Transfinite Reductions".

In addition to the contributed papers, the RTA program contained an invited talk by Mikolaj Bojanczyk on "Automata for Data Words and Data Trees" and by Vincent van Oostrom on "Realising Optimal Sharing". In addition, there were invited talks by David Harel, Gordon Plotkin, Georg Gottlob and J Strother Moore, jointly with ITP, LICS, RTA and SAT.

.

© C. Lynch
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.XIII

Maybe people helped to make RTA 2010 a success. I would like to thank Zhiqiang Liu for helping with the proceedings and Ralph Eric McGregor for helping with the RTA webpage. FLOC was hosted by the School of Informatics at the University of Edinburgh, Scotland. Support by the conference sponsors – EPSRC, NSF, Microsoft Research, Association for Symbolic Logic, CADE Inc., Google, Hewlett-Packard, Intel – is gratefully acknowledged.

June 2010 Christopher Lynch

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 1-4
http://rewriting.loria.fr/rta/

AUTOMATA FOR DATA WORDS AND DATA TREES

MIKOŁAJ BOJAŃCZYK

University of Warsaw
E-mail address: bojan@mimuw.edu.pl
URL: www.mimuw.edu.pl/∼bojan

Abstract. Data words and data trees appear in verification and XML processing. The term "data" means that positions of the word, or tree, are decorated with elements of an infinite set of data values, such as natural numbers or ASCII strings. This talk is a survey of the various automaton models that have been developed for data words and data trees.

A *data word* is a word where every position carries two pieces of information: a *label* from a finite alphabet, and a *data value* from an infinite set. A data tree is defined likewise.

As an example, suppose that the finite alphabet has two labels `request` and `grant`, and the data values are numbers (interpreted as process identifiers). A data word, such as the one below, can be seen as log of events that happened to the processes.

request	request	request	grant	request	grant	request	request	...
1	2	1	1	3	3	7	7	

The example with processes and logs can be used to illustrate how data words are used in verification. In one formulation, verification is a decision problem with two inputs: a correctness property, and a scheduling mechanism. The correctness property is some set K of "correct" logs. Often in verification, one talks about infinite words. For example, we might be interested in the following liveness property:

> "For every position with label `request`, there exists a later position with label `grant` and the same data value."

The scheduling mechanism is represented as the set L of logs consistent with the mechanism. For example, a rather unwise scheduler would result in the following logs:

> "Every position with label `grant` carries the same data value as the most recent position with label `request`."

The verification problem is the question: does the scheduling mechanism guarantee the correctness property? (In the example given here, the answer is no.) In terms of languages, this is the question if the difference $L - K$ is nonempty. To complete the description of the

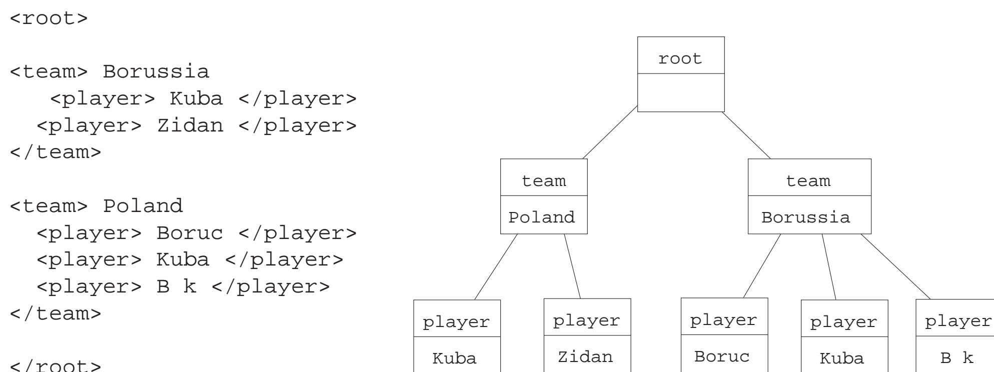
1998 ACM Subject Classification: PREFERRED list of ACM classifications.
Key words and phrases: Automata, Infinite State Systems.
Work supported by the FET programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.

© M. Bojańczyk
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.1

problem, one should indicate some way of representing the languages L and K; a typical solution is to use some variant of logic or automata.

Another application of data is in XML. Here, (finite) data trees are the pertinent object. The data values are used to model the text content in an XML document, while the labels are used to model tag names, as illustrated in the following example. The finite alphabet describes three possible tag names: `root`, `team` and `player`. The data values are ASCII strings that describe team names and player names. The picture below depicts an XML document and its interpretation as a data tree.



One might want to express correctness properties of an XML document, such as "no player is a member of two different teams". A typical algorithmic problem would concern the relation between two correctness properties, such as: does correctness property L imply correctness property K? This is a problem like the one in the verification example, although the logics used for describing XML documents usually have a different flavor than the logics for describing behavior of processes. Another algorithmic problem is to query documents, such as "find the nodes that describe a player who plays in two different teams". For querying, algorithms should be very fast, for instance linear in the document size. For verification, sometimes even decidability is hard to get.

The problems described above are well understood in the data-free setting, where positions carry only labels and not data values. Automata techniques have been highly successful in this area.

What about data? What is the right automaton model for data words and data trees? Recent years have seen a lot of work in this direction, with many incompatible definitions being proposed. Some of the approaches are listed in the references. Which one is the right one? What is a "regular language" in the presence of data? We do not know yet, and maybe we never will. It is difficult (indeed, impossible, under a certain formulation) to design an automaton model that is robust (the languages recognized have good closure properties, such as boolean operations and projections), expressive (captures some reasonable languages, such as "all positions with label a have the same data value") and decidable (e.g. has decidable emptiness). Undecidable problems, like the Post Embedding Problem, can be easily encoded in data words, in many different ways.

The talk surveys the varied landscape of automata models for data words and data trees. I will talk about the technical aspect of deciding emptiness, including the connection with vector addition systems (Petri Nets), as well as the connection with well-quasi-orders.

I will also talk about the technical aspect of efficient evaluation, including the connection with semigroup theory.

References

[1] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, pages 25–36, 2005.
[2] H. Björklund and T. Schwentick. On notions of regularity for data languages. In *FCT*, pages 88–99, 2007.
[3] Mikołaj Bojańczyk, Sławomir Lasota. An extension of data automata that captures XPath . To appear in *LICS*, 2010.
[4] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3), 2009.
[5] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.
[6] Mikołaj Bojańczyk, Paweł Parys. Efficient evaluation of nondeterministic automata using factorization forests. To appear in *ICALP*, 2010.
[7] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3), 2009.
[8] Diego Figueira. Forward-XPath and extended register automata on data-trees. In *ICDT*, 2010. To appear.
[9] Marcin Jurdziński and Ranko Lazić. Alternation-free modal mu-calculus for data trees. In *LICS*, pages 131–140, 2007.
[10] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
[11] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57, 2006.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 5-6
http://rewriting.loria.fr/rta/

REALISING OPTIMAL SHARING

VINCENT VAN OOSTROM

Department of Philosophy, Utrecht University
Heidelberglaan 6, 3584 CS Utrecht, The Netherlands
E-mail address: Vincent.vanOostrom@phil.uu.nl

1998 ACM Subject Classification: F.4.2.
Key words and phrases: Optimal Sharing.

© Vincent van Oostrom
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.5

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 7-16
http://rewriting.loria.fr/rta/

AUTOMATED CONFLUENCE PROOF BY DECREASING DIAGRAMS BASED ON RULE-LABELLING

TAKAHITO AOTO [1]

[1] RIEC, Tohoku University, 2-1-1 Katahira, Sendai, Miyagi, 980-8577, Japan
E-mail address: aoto@nue.riec.tohoku.ac.jp
URL: http://www.nue.riec.tohoku.ac.jp/user/aoto/

Abstract. Decreasing diagrams technique (van Oostrom, 1994) is a technique that can be widely applied to prove confluence of rewrite systems. To directly apply the decreasing diagrams technique to prove confluence of rewrite systems, rule-labelling heuristic has been proposed by van Oostrom (2008). We show how constraints for ensuring confluence of term rewriting systems constructed based on the rule-labelling heuristic are encoded as linear arithmetic constraints suitable for solving the satisfiability of them by external SMT solvers. We point out an additional constraint omitted in (van Oostrom, 2008) that is needed to guarantee the soundness of confluence proofs based on the rule-labelling heuristic extended to deal with non-right-linear rules. We also present several extensions of the rule-labelling heuristic by which the applicability of the technique is enlarged.

1. Introduction

Confluent term rewriting systems form a basis of flexible computations and effective deductions for equational reasoning [2, 10]. Thus, confluence is considered to be one of the most important properties for *term rewriting systems* (*TRSs* for short). In contrast to the termination proof techniques, where automation of the techniques has been actively investigated, not much attention has been paid to *automation of confluence proving*. Motivated by such a situation, Aoto et.al. [1, 15] have started developing a fully-automated confluence prover ACP.

Decreasing diagrams technique [11] is a technique that can be widely applied to prove confluence of rewrite systems. Many confluence results are explained and are extended based on the decreasing diagrams criterion [11, 13, 14]. In [13], *rule-labelling heuristic* has been proposed to prove confluence of rewrite systems directly by the decreasing diagrams technique. In the rule-labelling heuristic, each rewrite step is labeled by the rewrite rule employed in that rewrite step—then the existence of the suitable ordering on labels ensures the confluence of the TRSs. In [1, 15], the implementation of decreasing diagrams techniques in ACP was left as a future work.

1998 ACM Subject Classification: D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems; I.2.2 [Artificial Intelligence]: Automatic Programming.

Key words and phrases: Confluence, Decreasing Diagrams, Automation, Term Rewriting Systems.

© T. Aoto
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.7

In this paper, we report a method to incorporate the confluence proof by the decreasing diagrams based on the rule-labelling heuristic into the automated confluence provers such as the ACP. More precisely, we show how conditions for ensuring confluence of term rewriting systems constructed based on the rule-labelling heuristic are encoded as *linear arithmetic constraints* suitable for solving the satisfiability of them by *external SMT solvers* (SAT modulo theories where the linear arithmetic is employed for the underlying theory). Furthermore, we point out an additional condition omitted in [13] that is needed to guarantee the soundness of confluence proofs based on the rule-labelling heuristic. We also present several extensions of the rule-labelling heuristic by which the applicability of the technique is enlarged. All methods are implemented and experiments are reported.

The remainder of the paper is organized as follows. In Sec. 2, we briefly explain notions and notations used in this paper. In Sec. 3, we explain the decreasing diagrams technique and present an encoding of confluence criteria based on the basic version of the rule-labelling heuristic. In Sec. 4, we explain an extension of rule-labelling heuristic for left-linear TRSs. Here we point out the necessity of an additional condition omitted in [13]. In Sec. 5, we present the encoding of the criterion explained in Sec. 4 with a natural generalization. In section 6, we present two further flexibilities that can be added to the heuristic. Sec. 7 reports an implementation and experiments. Sec. 8 concludes.

2. Preliminaries

This section briefly explains notions and notations used in this paper. For omitted definitions, we refer [2].

Abstract reduction system (*ARS* for short) $\mathcal{A} = \langle A, (\to_i)_{i \in I} \rangle$ consists of a set A and indexed relations $\to_i$ over A. For $J \subseteq I$, $\to_J = \bigcup_{i \in J} \to_i$ and $\to_I$ is abbreviated to $\to$ if no confusion arises. The reverse of $\to$ is denoted by $\leftarrow$. The reflexive transitive closure (reflexive closure, equivalence closure) of $\to$ is denoted by $\xrightarrow{*}$ (resp. $\xrightarrow{=}$, $\xleftrightarrow{*}$). We use $\circ$ for the composition of relations. We denote a quasi-order by $\succsim$, its strict part by $\succ$ and its equivalence part by $\simeq$. A quasi-order is *well-founded* if there exists no infinite descending chain $a_0 \succ a_1 \succ \cdots$. We put $\prec m = \{i \mid i \prec m\}$, $\simeq m = \{i \mid i \simeq m\}$ and $\prec l, m = \{i \mid i \prec l\} \cup \{i \mid i \prec m\}$. The lexicographic comparison $\succsim_{\text{lex}}$ by two quasi-orders $\succsim_1$ and $\succsim_2$ is given by $\langle a_1, a_2 \rangle \succsim_{\text{lex}} \langle b_1, b_2 \rangle$ iff either $a_1 \succ_1 b_1$ or $a_1 \simeq_1 b_1$ and $a_2 \succsim_2 b_2$.

The sets of *function symbols* and *variables* are denoted by $\mathcal{F}$ and V. Each function symbol f is equipped with a natural number $\text{arity}(f)$, the *arity* of f. A *constant* is a function symbol with arity 0. We denote by $\text{V}(t)$ the set of variables occurring in a term t. A variable $x \in \text{V}(t)$ is said to have a *linear occurrence in* t (or x is *linear in* t) if there is only one occurrence of x in t. A term t is *linear* if all variables in $\text{V}(t)$ are linear in t. A *position* in a term is denoted by a (possibly empty) sequence of positive integers. The empty sequence (i.e. the *root* position) is denoted by ϵ. If p is a position in a term t, the *subterm* of t at p is denoted by t/p and we write $t[s]_p$ the term obtained from t by replacing the subterm at p with a term s. A *context* is a term with a special constant $\Box$ (called a *hole*). A context C with precisely one occurrence of the hole is denoted by $C[\,]$. We write $C[\,]_p$ if $C[\,]/p = \Box$. An instance of t by a substitution σ is written as $t\sigma$.

Any *rewrite rule* $l \to r$ satisfies the conditions (1) $l \notin V$ and (2) $\text{V}(r) \subseteq \text{V}(l)$. Rewrite rules are identified modulo variable renaming. A rewrite rule $l \to r$ is *linear* (*left-linear*, *right-linear*) if l and r (l, r, respectively) are linear. A *term rewriting system* (*TRS* for short) is a finite set of rewrite rules. It is linear (left-linear, right-linear) if so are all rules.

There is a *rewrite step* $s \to t$ if there exist a context $C[\,]_p$, a substitution σ, and a rewrite rule $l \to r \in \mathcal{R}$ such that $s = C[l\sigma]_p$ and $t = C[r\sigma]_p$. The subterm occurrence of $l\sigma$ at p is the *redex occurrence* of this rewrite step; the occurrence of l (except variables) in s is called the *redex pattern* of this rewrite step. A rewrite sequence of the form $s \leftarrow u \to t$ is called a *peak*; the one of the form $s \stackrel{*}{\to} \circ \stackrel{*}{\leftarrow} t$ is a *joinable rewrite sequence*. Terms s and t are *joinable* if $s \stackrel{*}{\to} \circ \stackrel{*}{\leftarrow} t$. A TRS $\mathcal{R}$ is *confluent* or *Church–Rosser* if $s \stackrel{*}{\leftarrow} \circ \stackrel{*}{\to} t$ implies s and t are joinable.

Let s, t be terms whose variables are disjoint. The term s *overlaps* on t (at position p) when there exists a non-variable subterm $u = t/p$ of t such that u and s are unifiable. Let $l_1 \to r_1$ and $l_2 \to r_2$ be rewrite rules w.l.o.g. whose variables are disjoint. Suppose that l_1 overlaps on l_2 at position p. Let σ be the most general unifier of l_1 and l_2/p. Then the term $l_2[l_1]\sigma$ yields a *critical peak* $l_2[r_1]\sigma \leftarrow l_2[l_1]\sigma \to r_2\sigma$. The pair $\langle l_2[r_1]\sigma, r_2\sigma\rangle$ is called the *critical pair* obtained by the overlap of $l_1 \to r_1$ on $l_2 \to r_2$ at position p. In the case of self-overlap (i.e. when $l_1 \to r_1$ and $l_2 \to r_2$ are identical modulo renaming), we do not consider the case $p = \epsilon$. The set of critical pairs obtained by an overlap of $l_1 \to r_1$ on $l_2 \to r_2$ is denoted by $\mathrm{CP}(l_1 \to r_1, l_2 \to r_2)$. The set of critical pairs in a TRS $\mathcal{R}$ is denoted by $\mathrm{CP}(\mathcal{R})$.

3. Confluence by decreasing diagrams based on rule-labelling heuristic

An ARS $\mathcal{A} = \langle A, (\to_i)_{i\in I}\rangle$ is *locally decreasing* w.r.t. a well-founded quasi-order $\succsim$ if for any $s \leftarrow_l \circ \to_m t$ there exists $s \stackrel{*}{\to}_{\prec l} \circ \stackrel{=}{\to}_{\simeq m} \circ \stackrel{*}{\to}_{\prec l,m} \circ \stackrel{*}{\leftarrow}_{\prec l,m} \circ \stackrel{=}{\leftarrow}_{\simeq l} \circ \stackrel{*}{\leftarrow}_{\prec m} t$ [11]. In this paper, we use the following variant of decreasing diagrams criterion.

Proposition 3.1 (decreasing diagrams criterion [11])**.** *An ARS $\mathcal{A} = \langle A, (\to_i)_{i\in I}\rangle$ is confluent if $\mathcal{A}$ is locally decreasing w.r.t. a well-founded quasi-order order $\succsim$.* ∎

In [13], *rule-labelling heuristic* is introduced to apply the decreasing diagrams criterion to directly prove confluence of TRSs. To use the decreasing diagrams criterion, each rewrite step needs to be equipped with a label—in the rule-labelling heuristic, the rewrite rule employed in the rewrite step is used as the label of each rewrite step. As in [13], let us suppose that each rewrite rule is numbered from 1 to $|\mathcal{R}|$ and that each rewrite rule is identified with its number ($i : l \to r \in \mathcal{R}$ indicates that i is the number of $l \to r$). We say a peak $s \leftarrow_i u \to_j t$ is *locally decreasing w.r.t.* $\succsim$ if there is a rewrite sequence of the form $s \stackrel{*}{\to}_{\prec i} \circ \stackrel{=}{\to}_{\simeq j} \circ \stackrel{*}{\to}_{\prec i,j} \circ \stackrel{*}{\leftarrow}_{\prec i,j} \circ \stackrel{=}{\leftarrow}_{\simeq i} \circ \stackrel{*}{\leftarrow}_{\prec j} t$.

Proposition 3.2 (confluence by rule-labelling heuristic [13])**.** *A linear TRS $\mathcal{R}$ is confluent if there exists a quasi-order $\succsim$ on $\mathcal{R}$ such that any critical peak of $\mathcal{R}$ is locally decreasing w.r.t. $\succsim$.* ∎

Note that well-founded of $\succsim$ follows from the finiteness of the set $\mathcal{R}$. Based on this proposition, a (basic) confluence proof of TRS $\mathcal{R}$ by decreasing diagrams based on rule-labelling is conducted as follows.

Step 1 Check (left- and right-)linearity.

Step 2 Find a joinable rewrite sequence $s \stackrel{*}{\to} v \stackrel{*}{\leftarrow} t$ for every critical pairs $\langle s, t\rangle \in \mathrm{CP}(i, j)$.

Step 3 Check whether there exists a quasi-order $\succsim$ on $\mathcal{R}$ such that $s \stackrel{*}{\to} v \stackrel{*}{\leftarrow} t$ (obtained in the step 2) has the form $s \stackrel{*}{\to}_{\prec i} \circ \stackrel{=}{\to}_{\simeq j} \circ \stackrel{*}{\to}_{\prec i,j} \circ \stackrel{*}{\leftarrow}_{\prec i,j} \circ \stackrel{=}{\leftarrow}_{\simeq i} \circ \stackrel{*}{\leftarrow}_{\prec j} t$ for every critical pairs $\langle s, t\rangle \in \mathrm{CP}(i, j)$.

Computation of the step 1 is easy. Automation of the step 2 is partially achieved by imposing a maximum length on rewrite steps $s \xrightarrow{*} v \xleftarrow{*} t$. The only non-trivial part is the step 3. We show that the step 3 can be solved by reducing the problem into the satisfiability of an arithmetic constraint.

First, let us illustrate by an example how the requirement on the quasi-order $\precsim$ is specified. Since the requirements on $s \xrightarrow{*} v$ and $v \xleftarrow{*} t$ are symmetric, we concentrate on the $s \xrightarrow{*} v$ part. Suppose that we have a joinable rewrite sequence $s \to_{x_1} \circ \to_{x_2} \circ \to_{x_3} v \xleftarrow{*} t$ for $\langle s,t \rangle \in \mathrm{CP}(i,j)$. The requirement is that $\to_{x_1} \circ \to_{x_2} \circ \to_{x_3}$ has the form $\xrightarrow{*}_{\prec i} \circ \xrightarrow{=}_{\simeq j} \circ \xrightarrow{*}_{\prec i,j}$. We can think of five possibilities depending on where the rewrite step equivalent to j (virtually) appears:

(i) $x_1, x_2, x_3 \in \prec i, j$ (i.e. the rewrite step equivalent to j is placed before $\to_{x_1}$ step)
(ii) $x_1 \simeq j$ and $x_2, x_3 \in \prec i, j$ (i.e. the rewrite step equivalent to j is $\to_{x_1}$ step)
(iii) $x_1 \prec i$, $x_2 \simeq j$ and $x_3 \in \prec i, j$ (i.e. the rewrite step equivalent to j is $\to_{x_2}$ step)
(iv) $x_1, x_2 \prec i$, $x_3 \simeq j$ (i.e. the rewrite step equivalent to j is $\to_{x_3}$ step)
(v) $x_1, x_2, x_3 \prec i$ (i.e. the rewrite step equivalent to j is placed after $\to_{x_3}$ step)

The last possibility is redundant because of the first one; thus four possibilities remain. Since $x_k \in \prec i, j$ equals to $(x_k \prec i) \vee (x_k \prec j)$, we obtain the following requirement on the quasi-order $\precsim$.

$$
\begin{array}{lll}
 & (x_1 \prec i \vee x_1 \prec j) \wedge (x_2 \prec i \vee x_2 \prec j) \wedge (x_3 \prec i \vee x_3 \prec j) & \text{from the case (i)} \\
\vee & (x_1 \simeq j) \wedge (x_2 \prec i \vee x_2 \prec j) \wedge (x_3 \prec i \vee x_3 \prec j) & \text{from the case (ii)} \\
\vee & (x_1 \prec i) \wedge (x_2 \simeq j) \wedge (x_3 \prec i \vee x_3 \prec j) & \text{from the case (iii)} \\
\vee & (x_1 \prec i) \wedge (x_2 \prec i) \wedge (x_3 \simeq j) & \text{from the case (iv)}
\end{array}
$$

In describing the requirement for the general case, the following assumption is assumed. Below, each rewrite sequence is supposed to be assigned by a sequence of labels as $s \xrightarrow{*}_{i_1 \cdots i_k} t$ for $s \to_{i_1} \circ \cdots \circ \to_{i_k} t$.

Assumption 3.3. *We assume that there exists a joinable rewrite sequence $s \xrightarrow{*}_\sigma \circ \xleftarrow{*}_\rho t$ for each critical pair $\langle s,t \rangle \in \mathrm{CP}(i,j)$. Given labelling of rewrite steps, the sequences σ and ρ of labels are denoted by $\mathrm{JL}(s,t)$ and $\mathrm{JR}(s,t)$, respectively.*

Definition 3.4. We define $\mathrm{Ldd}(i,j,x_1 \cdots x_n)$ and $\mathrm{LDD}(\mathcal{R})$ as follows.

$$
\begin{array}{l}
\mathrm{Ldd}(i,j,x_1 \cdots x_n) = (\bigwedge_{1 \le l \le n}((x_l \prec i) \vee (x_l \prec j))) \vee \\
\qquad \bigvee_{1 \le k \le n} \left[(\bigwedge_{1 \le l < k}(x_l \prec i)) \wedge (x_k \simeq j) \wedge (\bigwedge_{k < l \le n}((x_l \prec i) \vee (x_l \prec j))) \right] \\
\mathrm{LDD}(\mathcal{R}) = \bigwedge_{i,j \in \mathcal{R}} \bigwedge_{\langle s,t \rangle \in \mathrm{CP}(i,j)} \big(\mathrm{Ldd}(i,j,\mathrm{JL}(s,t)) \wedge \mathrm{Ldd}(j,i,\mathrm{JR}(s,t)) \big)
\end{array}
$$

Theorem 3.5. *A linear TRS $\mathcal{R}$ is confluent if there exists a quasi-order $\succsim$ on $\mathcal{R}$ that satisfies* $\mathrm{LDD}(\mathcal{R})$. ∎

We next explain how the existence problem of a quasi-order $\succsim$ on $\mathcal{R}$ that satisfy $\mathrm{LDD}(\mathcal{R})$ is reduced to the satisfiability problem of an arithmetic constraint. The idea is to specify the quasi-order $\succsim$ by the assignment of natural number weights, that is, $i \succ j$ iff the rule i has the weight strictly larger than that of j. Here, note that since $\mathcal{R}$ is finite and the requirement is a monotone formula, it suffices to consider the total quasi-order. Suppose non-negative integer variables $w_1, \ldots, w_{|\mathcal{R}|}$ are to be assigned by the weight of the rules $1, \ldots, |\mathcal{R}| \in \mathcal{R}$. Then the requirement $\mathrm{LDD}(\mathcal{R})$ of $\succsim$ is translated to an arithmetic constraint $[\![\mathrm{LDD}]\!](\mathcal{R})$ over the indeterminates $w_1, \ldots, w_{|\mathcal{R}|}$ and the existence problem of a quasi-order $\succsim$ satisfying

LDD($\mathcal{R}$) is reduced to the existence problem of a suitable assignment on indeterminates $w_1, \ldots, w_{|\mathcal{R}|}$ which satisfies $[\![\mathrm{LDD}]\!](\mathcal{R})$.

Definition 3.6. An arithmetic constraint $[\![\mathrm{LDD}]\!](\mathcal{R})$ is defined as follows.

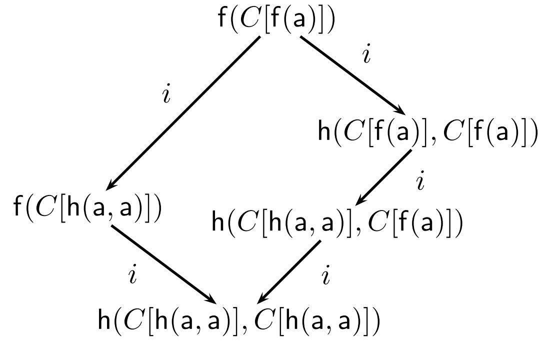
$$[\![\mathrm{Ldd}]\!](i,j,x_1\cdots x_n) = (\bigwedge_{1\le l\le n}((w_{x_l} < w_i) \vee (w_{x_l} < w_j))) \vee$$
$$\bigvee_{1\le k\le n}\left[(\bigwedge_{1\le l<k}(w_{x_l} < w_i)) \wedge (w_{x_k} = w_j) \wedge (\bigwedge_{k<l\le n}((w_{x_l} < w_i) \vee (w_{x_l} < w_j)))\right]$$
$$[\![\mathrm{LDD}]\!](\mathcal{R}) = \bigwedge_{i,j\in\mathcal{R}} \bigwedge_{\langle s,t\rangle\in \mathrm{CP}(i,j)} \left([\![\mathrm{Ldd}]\!](i,j,\mathrm{JL}(s,t)) \wedge [\![\mathrm{Ldd}]\!](j,i,\mathrm{JR}(s,t))\right)$$

Theorem 3.7. *A linear TRS $\mathcal{R}$ is confluent if $[\![\mathrm{LDD}]\!](\mathcal{R})$ is satisfiable.* ∎

Since constrains $[\![\mathrm{LDD}]\!](\mathcal{R})$ is a boolean combination of linear arithmetic formulas (every monomial contains only one variable), the satisfiability of $[\![\mathrm{LDD}]\!](\mathcal{R})$ is efficiently checked by an external SMT (SAT modulo theories) solver where the linear arithmetic is employed for the underlying theory.

4. Rule-labelling heuristic capable of non-right-linear rules

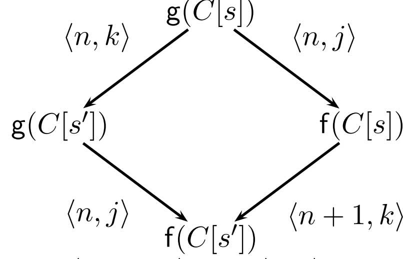
As explained in [13], the rule-labelling heuristic is not applicable to non-linear TRSs, but by adding some additional information to labels, the technique can be extended to handle (possibly non-right-linear) left-linear TRSs. To explain this extension, let us replicate a situation from Example 20 of [13]. Suppose $i : \mathsf{f}(x) \to \mathsf{h}(x,x) \in \mathcal{R}$. To apply the decreasing diagrams criterion (Proposition 3.1), one has to impose the local decreasingness for peaks arising from nested overlaps of the same rewrite rule i such as:



This peak, however, is not locally decreasing as $\to_i \circ \to_i$ does not have the form $\xrightarrow{*}_{\prec i} \circ \xrightarrow{=}_{\simeq i} \circ \xrightarrow{*}_{\prec i,i}$. Hence the rule-labelling heuristic fails.

An idea to solve this situation is to extend the label i to $\langle m, i\rangle$ where m is the number of occurrences of f on the path from the redex to the root and use the lexicographic comparison (denoted by $\succsim_{\mathrm{lex}}$) in which the first component is compared with the usual ordering $\geq$ on natural numbers [13]. Then we have labeled rewrite steps $\mathsf{f}(C[\mathsf{f}(\mathsf{a})]) \to_{\langle n+1,i\rangle} \mathsf{f}(C[\mathsf{h}(\mathsf{a},\mathsf{a})])$ and $\mathsf{h}(C[\mathsf{f}(\mathsf{a})], C[\mathsf{f}(\mathsf{a})]) \to_{\langle n,i\rangle} \mathsf{h}(C[\mathsf{h}(\mathsf{a},\mathsf{a})], C[\mathsf{f}(\mathsf{a})]) \to_{\langle n,i\rangle} \mathsf{h}(C[\mathsf{h}(\mathsf{a},\mathsf{a})], C[\mathsf{f}(\mathsf{a})])$, provided that the context $C[\,]$ has n-occurrences of f on the path from the hole to the root. Then, by $\langle n+1, i\rangle \succ_{\mathrm{lex}} \langle n, i\rangle$, the local decreasingness of the peak is ensured.

Although it is not mentioned in [13], one should note that this extended heuristic does not work if there is a rewrite rule such as $j : \mathsf{g}(x) \to \mathsf{f}(x)$ in $\mathcal{R}$ whose rewrite step may increase the number of occurrences of f above a redex. For example, a critical peak below arising from the nested overlap of redex patterns is not locally decreasing:



as $\langle n+1,k\rangle \succ_{\rm lex} \langle n,k\rangle$ and $\langle n+1,k\rangle \succ_{\rm lex} \langle n,j\rangle$.

To avoid such a situation, one needs to impose an additional condition: for any contexts $C_l[\,], C_r[\,]$ and $x \in V$ such that $C_l[x] \to C_r[x] \in \mathcal{R}$,

- $\sharp_{\mathsf{f}} C_l[\,] \geq \sharp_{\mathsf{f}} C_r[\,]$ if x is linear in $C_r[x]$, and
- $\sharp_{\mathsf{f}} C_l[\,] > \sharp_{\mathsf{f}} C_r[\,]$ if x is not linear in $C_r[x]$

where $\sharp_{\mathsf{f}}\, C[\,]$ denotes the number of occurrences of the function symbol f along the path from the hole to the root in the context $C[\,]$.

5. A generalization of rule-labelling heuristic for left-linear TRSs

In this section, we extend the encoding presented in Sec. 3 to the rule-labelling heuristic for left-linear TRSs explained in Sec. 4 under the following natural generalization:

(1) Not only f but some subset $\mathcal{G}$ of function symbols can be designated for counting occurrences (on the path from the redex to the root).
(2) More generally, the counting of occurrences can be generalized to the summation of weights of ≥ 0 assigned for each function symbol's occurrences.

The summation of weights is formalized by a notion of the weight of context.

Definition 5.1. Let $C[\,]$ be a context and $w : \mathcal{F} \to \mathbb{N}$ be a function where $\mathbb{N}$ is the set of natural numbers. The *weight* $\sharp C[\,]$ of a context $C[\,]$ is defined as follows.

$$\sharp C[\,] = \begin{cases} 0 & \text{if } C[\,] = \Box \\ w(f) + \sharp \tilde{C}[\,] & \text{if } C[\,] = f(\ldots, \tilde{C}[\,], \ldots) \end{cases}$$

To encode the weight $\sharp C[\,]$, we introduce a non-negative integer variable z_f for each $f \in \mathcal{F}$ to be assigned by $w(f)$. Then $\sharp C[\,]$ is encoded by a polynomial $[\![\sharp C[\,]]\!]$ whose definition is obtained by replacing $w(f)$ by z_f in the definition of $\sharp C[\,]$. Thus the label of each rewrite step is encoded by $\langle \varphi, x\rangle$ where $x \in \{1,\ldots,|\mathcal{R}|\}$ and φ is a polynomial over indeterminate $(z_f)_{f\in\mathcal{F}}$. We assume that $\mathrm{JL}(s,t)$ and $\mathrm{JR}(s,t)$ are updated accordingly. The set $\mathrm{CP}_2(i,j)$ of critical pairs equipped with the weight of peak rewrite steps is given like this: $\mathrm{CP}_2(i,j) = \{\big\langle \langle [\![\sharp l_j[\,]_p\sigma]\!], s\rangle, \langle 0,t\rangle \big\rangle \mid s = l_j[r_i]_p\sigma \leftarrow l_j[l_i]_p\sigma = l_j\sigma \to r_j\sigma = t, \langle s,t\rangle \in \mathrm{CP}(i,j)\}$.

Definition 5.2. Arithmetic constraints $[\![\mathrm{LDD}_2]\!](\mathcal{R})$ and $[\![\mathrm{CND}]\!](\mathcal{R})$ are defined as follows.

$$\langle \varphi, i\rangle \prec_{\rm lex} \langle \rho, j\rangle = (\varphi < \rho \lor (\varphi = \rho \land w_i < w_j)) \qquad \langle \varphi, i\rangle \simeq_{\rm lex} \langle \rho, j\rangle = (\varphi = \rho \land w_i < w_j)$$

$$[\![\mathrm{Ldd}_2]\!](\vec{\varphi},\vec{\psi},\vec{\rho}_1\cdots\vec{\rho}_n) = (\textstyle\bigwedge_{1\leq l\leq n}((\vec{\rho}_l \prec_{\rm lex} \vec{\varphi}) \lor (\vec{\rho}_l \prec_{\rm lex} \vec{\psi}))) \lor \\ \textstyle\bigvee_{1\leq k\leq n} \big[(\bigwedge_{1\leq l<k}(\vec{\rho}_l \prec_{\rm lex} \vec{\varphi})) \land (\vec{\rho}_k \simeq_{\rm lex} \vec{\psi}) \land (\bigwedge_{k<l\leq n}((\vec{\rho}_l \prec_{\rm lex} \vec{\varphi}) \lor (\vec{\rho}_l \prec_{\rm lex} \vec{\psi})))\big]$$

$$[\![\mathrm{LDD}_2]\!](\mathcal{R}) = \textstyle\bigwedge_{i,j\in\mathcal{R}} \bigwedge_{\langle\langle\varphi,s\rangle,\langle\psi,t\rangle\rangle\in\mathrm{CP}_2(i,j)} \big([\![\mathrm{Ldd}_2]\!](\langle\varphi,i\rangle,\langle\psi,j\rangle,\mathrm{JL}(s,t)) \\ \land [\![\mathrm{Ldd}_2]\!](\langle\psi,j\rangle,\langle\varphi,i\rangle,\mathrm{JR}(s,t))\big)$$

$$[\![\mathrm{CND}]\!](\mathcal{R}) = \bigwedge\{[\![\sharp C_l[\,]]\!] \geq [\![\sharp C_r[\,]]\!] \mid C_l[x] \to C_r[x] \in \mathcal{R}, x \notin \mathrm{V}(C_r[\,])\} \\ \wedge \bigwedge\{[\![\sharp C_l[\,]]\!] > [\![\sharp C_r[\,]]\!] \mid C_l[x] \to C_r[x] \in \mathcal{R}, x \in \mathrm{V}(C_r[\,])\}$$

We here explain the constraint $[\![\mathrm{CND}]\!](\mathcal{R})$ by an example.

Example 5.3. Let $\mathcal{R} = \{\ \mathsf{f}(\mathsf{g}(x), y) \to \mathsf{h}(x, \mathsf{f}(x, y))\ \}$. Then the condition for variable y which is linear in the right-hand side (rhs for short) of the rule is encoded like this:

$$(1)\ z_{\mathsf{f}} \ \geq\ z_{\mathsf{h}} + z_{\mathsf{f}} \qquad (\text{for } C_l[\,] = \mathsf{f}(\mathsf{g}(x), \Box), C_r[\,] = \mathsf{h}(x, \mathsf{f}(x, \Box))).$$

The condition for variable x which is non-linear in rhs of the rule is encoded like this:

$$(2)\ z_{\mathsf{f}} + z_{\mathsf{g}} \ >\ z_{\mathsf{h}} \qquad (\text{for } C_l[\,] = \mathsf{f}(\mathsf{g}(\Box), y), C_r[\,] = \mathsf{h}(\Box, \mathsf{f}(x, y))),$$
$$(3)\ z_{\mathsf{f}} + z_{\mathsf{g}} \ >\ z_{\mathsf{h}} + z_{\mathsf{f}} \qquad (\text{for } C_l[\,] = \mathsf{f}(\mathsf{g}(\Box), y), C_r[\,] = \mathsf{h}(x, \mathsf{f}(\Box, y))).$$

Therefore $[\![\mathrm{CND}]\!](\mathcal{R}) = (1) \wedge (2) \wedge (3)$.

Theorem 5.4. *A left-linear TRS $\mathcal{R}$ is confluent if $[\![\mathrm{LDD}_2]\!](\mathcal{R}) \wedge [\![\mathrm{CND}]\!](\mathcal{R})$ is satisfiable.* ∎

Example 5.5. The following TRS $\mathcal{R}_1$ is from [4] (Example 20 of [14]).

$$\mathcal{R}_1 = \left\{ \begin{array}{llllllllllll} (1) & \mathsf{g}(\mathsf{a}) & \to & \mathsf{f}(\mathsf{g}(\mathsf{a})) & (3) & \mathsf{a} & \to & \mathsf{b} & (5) & \mathsf{h}(x,y) & \to & \mathsf{c} \\ (2) & \mathsf{g}(\mathsf{b}) & \to & \mathsf{c} & (4) & \mathsf{f}(x) & \to & \mathsf{h}(x,x) & & & & \end{array} \right\}$$

There is a (unique) critical peak of $\mathcal{R}$: $\mathsf{g}(\mathsf{b}) \leftarrow_{\langle z_{\mathsf{g}}, w_3\rangle} \mathsf{g}(\mathsf{a}) \to_{\langle 0, w_1\rangle} \mathsf{f}(\mathsf{g}(\mathsf{a}))$, which is joinable as $\mathsf{g}(\mathsf{b}) \to_{\langle 0, w_2\rangle} c \leftarrow_{\langle 0, w_5\rangle} \mathsf{h}(\mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a})) \leftarrow_{\langle 0, w_4\rangle} \mathsf{f}(\mathsf{g}(\mathsf{a}))$. By solving the constraint, a solution $z_{\mathsf{f}} = w_i = 1$ ($i \in \{1,3,5\}$), $z_f = w_2 = w_4 = 0$ ($f \in \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{g},\mathsf{h}\}$) is obtained.

Example 5.6. Let $\mathcal{R}_1$ be the TRS given in Example 5.5 and $\mathcal{R}_2 = \mathcal{R}_1 \cup \{\mathsf{g}(x) \to \mathsf{f}(\mathsf{f}(x))\}$. The weight assignment in Example 5.5 does not work for $\mathcal{R}_2$ because of $[\![\mathrm{CND}]\!](\mathcal{R}_2)$. In fact, the generated constraint $[\![\mathrm{LDD}_2]\!](\mathcal{R}_2) \wedge [\![\mathrm{CND}]\!](\mathcal{R}_2)$ is not satisfiable if we limit $z_f \in \{0,1\}$. By solving the constraint, a solution $z_{\mathsf{f}} = w_i = 1$ ($i \in \{1,5,6\}$), $z_{\mathsf{g}} = w_2 = 2$, $w_3 = 3$, $z_f = w_5 = 0$ ($f \in \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{g},\mathsf{h}\}$) is obtained. Thus one concludes $\mathcal{R}_2$ is confluent. This example demonstrates that our generalization from counting of function symbol's occurrences to summation of weight properly extends the applicability of the rule-labelling heuristic.

6. Adding further flexibilities to the rule-labelling heuristic

In this section, we add two further flexibilities to the rule-labelling heuristic.

6.1. Adding flexibility on the lexicographic comparison

In the previous section (and also in [13]) the label $\langle \sharp C[\,], i\rangle$ is compared in such a way that first on the weight $\sharp C[\,]$ and then on the weight of the rule i. It is easy to see, however, that comparing the components in the reverse order can be used either.

Example 6.1. Let

$$\mathcal{R}_3 = \left\{ \begin{array}{llllllllllll} (1) & \mathsf{c} & \to & \mathsf{f}(\mathsf{a}) & (3) & \mathsf{a} & \to & \mathsf{g}(\mathsf{a}) & (5) & \mathsf{f}(x) & \to & \mathsf{h}(x,x) \\ (2) & \mathsf{c} & \to & \mathsf{f}(\mathsf{b}) & (4) & \mathsf{b} & \to & \mathsf{g}(\mathsf{g}(\mathsf{a})) & & & & \end{array} \right\}.$$

First note that in $\mathcal{R}_3$ $z_{\mathsf{f}} > z_{\mathsf{h}}$ need to be satisfied by the rule (5). Consider the critical peak $\mathsf{f}(\mathsf{a}) \leftarrow_{\langle 0, w_1\rangle} \mathsf{c} \to_{\langle 0, w_2\rangle} \mathsf{f}(\mathsf{b})$ and a joinable rewrite sequence $\mathsf{f}(\mathsf{a}) \to_{\langle z_{\mathsf{f}}, w_3\rangle} \mathsf{f}(\mathsf{g}(\mathsf{a})) \to_{\langle z_{\mathsf{f}}+z_{\mathsf{g}}, w_3\rangle} \mathsf{f}(\mathsf{g}(\mathsf{g}(\mathsf{a}))) \leftarrow_{\langle z_{\mathsf{f}}, w_4\rangle} \mathsf{f}(\mathsf{b})$ for it. As z_{f} is positive, there is no chance to satisfy local decreasingness condition if the comparison by $\langle \sharp C[\,], i\rangle$ is used. On the other hand, if one uses the comparison by $\langle i, \sharp C[\,]\rangle$, a suitable assignment is found.

Another workaround here is to consider another joinable rewrite sequence with auxiliary (duplicating) rewrite steps by the rule (5): $\mathsf{f}(\mathsf{a}) \to \mathsf{h}(\mathsf{a},\mathsf{a}) \xrightarrow{*} \circ \xleftarrow{*} \mathsf{h}(\mathsf{b},\mathsf{b}) \leftarrow \mathsf{f}(\mathsf{b})$. For this, however, it is required to search a joinable rewrite sequence with non-minimal length.

Benefit from both ways of comparison is obtained easy—it suffices to prepare new integer variables $w'_1,\ldots,w'_{|\mathcal{R}|}$ and change the encoding $\langle w_i, \varphi\rangle$ to $\langle w_i, \varphi, w'_i\rangle$, where the third component is used to encode the secondary quasi-order on rules compared after the comparison of the context weight φ.

6.2. Adding flexibility on the weight function

It is also easy to see that the weight function $\sharp$ for the context can be changed in such a way that the counted weight on an occurrence of the same function symbol is changed according to the argument position containing the hole.

Definition 6.2. Let $C[\,]$ be a context and $w : \mathcal{F}_{\mathbb{N}} \to \mathbb{N}$ be a function, where $\mathcal{F}_{\mathbb{N}} = \{\langle f, i\rangle \mid f \in \mathcal{F}, 1 \le i \le \mathrm{arity}(f)\}$. The weight $\sharp' C[\,]$ of the context $C[\,]$ is defined as follows.

$$\sharp' C[\,] = \begin{cases} 0 & \text{if } C[\,] = \Box \\ w(f,i) + \sharp'\tilde{C}[\,] & \text{if } C[\,] = f(\ldots,\tilde{C}[],\ldots) \text{ and } C[\,]/i = \tilde{C}[\,] \end{cases}$$

To encode the weight $\sharp' C[\,]$, non-negative integer variables $(z_{f,i})_{\langle f,i\rangle \in \mathcal{F}_{\mathbb{N}}}$ are introduced. Using the weight function $\sharp'$ rather than $\sharp$ is sometimes advantageous as witnessed in the following example.

Example 6.3. Let

$$\mathcal{R}_4 = \left\{ \begin{array}{llcl llcl} (1) & \mathsf{f}(\mathsf{f}(x,y),z) & \to & \mathsf{f}(x,\mathsf{f}(y,z)) & (3) & \mathsf{f}(x,\mathsf{1}) & \to & \mathsf{f}(\mathsf{1},x) \\ (2) & \mathsf{f}(\mathsf{1},x) & \to & x & & & & \end{array} \right\}.$$

For a critical peak $\mathsf{f}(\mathsf{f}(w,\mathsf{f}(x,y)),z)) \leftarrow_{\langle z_{\mathsf{f},1},w_1\rangle} \mathsf{f}(\mathsf{f}(\mathsf{f}(w,x),y),z) \to_{\langle 0,w_1\rangle} \mathsf{f}(\mathsf{f}(w,x),\mathsf{f}(y,z)))$, there is a joinable rewrite sequence $\mathsf{f}(\mathsf{f}(w,\mathsf{f}(x,y)),z)) \to_{\langle 0,w_1\rangle} \mathsf{f}(w,\mathsf{f}(\mathsf{f}(x,y),z)) \to_{\langle z_{\mathsf{f},2},w_1\rangle} \mathsf{f}(w,\mathsf{f}(x,\mathsf{f}(y,z))) \leftarrow_{\langle 0,w_1\rangle} \mathsf{f}(\mathsf{f}(w,x),\mathsf{f}(y,z)))$. It is readily convinced that the diagram can not be made locally decreasing unless we distinguish $z_{\mathsf{f},1}$ and $z_{\mathsf{f},2}$.

7. Implementation and experiments

All techniques described in this paper have been implemented. The implementation is written in SML/NJ[1] and built upon the confluence prover ACP. We have used Yices[2] [3] as an external SMT solver. In searching of a joinable rewrite sequence of critical pairs, the following heuristics are employed: (i) set the maximum number of rewrite steps to 5. (ii) joinability is tested from reducts obtained in smaller steps (all joinable sequences obtained in the smallest step are considered but not any others with larger steps.)

We have tested various versions of rule-labelling heuristic described in this paper. The summary of experiments is described in Table 1. (1)–(5) are results of confluence proofs by decreasing diagrams based on the rule-labelling heuristics. We have also presented results of other confluence proving techniques for left-linear TRSs for comparison. The columns below the title $\mathcal{R}_i$ show success (✓) or failure (×) of the proof attempts to TRSs $\mathcal{R}_1$–$\mathcal{R}_4$

[1] `http://www.smlnj.org/`

[2] `http://yices.csl.sri.com/`

	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$	$\mathcal{R}_4$	*Col.*(msec)
Decreasing diagrams technique based on rule-labelling					
(1) basic version (Thm. 3.7)	×	×	×	×	35 (200)
(2) counting designated function symbol's occurrences	✓	×	×	×	41 (486)
(3) with context weight (Thm. 5.4)	✓	✓	×	×	41 (481)
(4) (3) + extended comparison (Subsect. 6.1)	✓	✓	✓	×	41 (795)
(5) (4) + extended context weight (Subsect. 6.2)	✓	✓	✓	✓	42 (692)
Other techniques for left-linear TRSs					
development closed TRSs [12]	×	×	×	×	16 (52)
linear strongly closed TRSs [6]	×	×	×	×	24 (52)
criterion by parallel critical pairs [9]	×	×	×	×	31 (58)
criterion by simultaneous critical pairs [7]	×	×	×	×	36 (91)
upside-parallel-closed/outside-closed TRSs [8]	×	×	×	×	19 (53)
All techniques	✓	✓	✓	✓	48 (593)
All techniques except the decreasing diagrams technique	×	×	×	×	40 (84)

Table 1: A summary of experiments

in the present paper. The columns below the title *Col.* show the number of success tested on a 106 collection of TRSs taken from various confluence-related papers and running time (msec.). All experiments have been performed on a FreeBSD platform of a PC equipped with 1.2GHz CPU and 1GB memory.

While other five techniques for left-linear TRSs proves 16–36 examples, the decreasing diagrams technique based on rule-labelling proves 45 examples ($\mathcal{R}_1$ is contained in the collection). Thus the comparison experimentally reveals the virtue of decreasing diagrams technique based on rule-labelling. The very basic version of the decreasing diagrams technique based on rule-labelling for linear TRSs already proves nearly 80% of the examples that can be proved with other extensions. Results on TRSs $\mathcal{R}_2$–$\mathcal{R}_4$ show that the refinements presented in the paper improve the applicability of the technique. The running time for decreasing diagrams technique based on rule-labelling is about 7–14 times larger than other five techniques. Since 34 examples are proved both in the decreasing diagrams technique based on rule-labelling and in the combination of other five techniques, it is better to try the other five techniques before the decreasing diagrams technique based on rule-labelling.

A new version of the confluence prover ACP involving all the techniques presented in the paper and the details of all experiments can be found on the webpage[3] of ACP.

8. Conclusion

We have described a method to automate confluence proofs by the decreasing diagrams based on the rule-labelling heuristic. We have shown an encoding of the confluence criterion into that of a linear arithmetic problem suitable for solving by external SMT solvers. An additional condition which need to be considered to guarantee the soundness of the technique (omitted in the original description of the heuristic [13]) and several generalizations of the heuristic which enlarge the applicability of the technique have been described. The presented technique has been implemented and the experiments show the advantage of incorporating the technique into automated confluence provers.

[3] `http://www.nue.riec.tohoku.ac.jp/tools/acp/`

Automation of decreasing diagram technique based on rule-labelling heuristic for linear TRSs has been obtained in [5] independently. Automation of the extended heuristic for left-linear TRSs, however, has not been explored in their paper. Instead, they are developing a new technique based on relative termination there.

In [13], another technique called self-duplication heuristic is described to deal with rule-labelling for (possibly non-right-linear) left-linear TRSs. In self-duplication heuristic, instead of counting function symbols' occurrences, parallel rewrite steps are considered to make critical peaks arising form nested overlaps of the non-right-linear rules locally decreasing. Automation of the decreasing diagrams technique with self-duplication heuristic remains as a future work.

Acknowledgments

The author thanks Yoshihito Toyama, Nao Hirokawa, Dominik Klein and anonymous referees for their helpful comments. This work was partially supported by a grant from JSPS No. 20500002.

References

[1] T. Aoto, Y. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. of RTA 2009*, *LNCS*, vol. 5595, pp. 93–102. Springer-Verlag, 2009.

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

[3] B. Dutertre and L. de Moura. The YICES SMT solver. Available from `http://yices.csl.sri.com/tool-paper.pdf`.

[4] B. Gramlich and S. Lucas. Generalizing Newman's lemma for left-linear rewrite systems. In *Proc. of RTA 2006*, *LNCS*, vol. 4098, pp. 187–201. Springer-Verlag, 2006.

[5] N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. *Computing Research Repository*, 0910.2853, 2009. Unpublished manuscript.

[6] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[7] S. Okui. Simultaneous critical pairs and Church-Rosser property. In *Proc. of RTA-98*, *LNCS*, vol. 1379, pp. 2–16. Springer-Verlag, 1998.

[8] M. Oyamaguchi and Y. Ohta. On the open problems concerning Church-Rosser of left-linear term rewriting systems. *IEICE Trans. Information and Systems*, E87-D(2):290–298, 2004.

[9] Y. Toyama. On the Church-Rosser property of term rewriting systems. Technical Report 17672, NTT ECL, 1981.

[10] Y. Toyama. Confluent term rewriting systems (invited talk). In *Proc. of RTA 2005*, *LNCS*, vol. 3467, p. 1. Springer-Verlag, 2005. Slides are available from `http://www.nue.riec.tohoku.ac.jp/user/toyama/slides/toyama-RTA05.pdf`.

[11] V. van Oostrom. Confluence by decreasing diagrams. *Theoretical Computer Science*, 126(2):259–280, 1994.

[12] V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[13] V. van Oostrom. Confluence by decreasing diagrams: converted. In *Proc. of RTA 2008*, *LNCS*, vol. 5117, pp. 306–320. Springer-Verlag, 2008.

[14] V. van Oostrom. Modularity of confluence: constructed. In *Proc. of IJCAR 2008*, *LNCS*, vol. 5195, pp. 348–363. Springer-Verlag, 2008.

[15] J. Yoshida, T. Aoto, and Y. Toyama. Automating confluence check of term rewriting systems. *Computer Software*, 26(2):76–92, 2009. In Japanese.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 17-32
http://rewriting.loria.fr/rta/

HIGHER-ORDER (NON-)MODULARITY

CLAUS APPEL [1] AND VINCENT VAN OOSTROM [2] AND JAKOB GRUE SIMONSEN [1]

[1] Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, 2100 Copenhagen Ø
Denmark
E-mail address, C. Appel: spectrum@diku.dk
E-mail address, J. G. Simonsen: simonsen@diku.dk

[2] ZENO Research Institute, Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht
The Netherlands
E-mail address: Vincent.vanOostrom@phil.uu.nl

ABSTRACT. We show that, contrary to the situation in first-order term rewriting, almost none of the usual properties of rewriting are modular for higher-order rewriting, irrespective of the higher-order rewriting format. We show that for the particular format of simply typed applicative term rewriting systems modularity of confluence, normalization, and termination can be recovered by imposing suitable linearity constraints.

1. Introduction and summary of results

The *disjoint union* of two rewrite systems is the rewrite system obtained by taking the disjoint union of their signatures and taking the union of their respective sets of rules. *Modularity* is the study of properties preserved and reflected when taking the disjoint union of two rewrite systems and has been studied intensely for first-order term rewriting systems.

Higher-order term rewriting adds two features to first-order term rewriting: meta-variables and binding. Meta-variables are variables for functions, i.e. they can be *applied*. Binding allows to construct functions by means of *abstraction*. There are a plethora of formats of higher-order rewriting, spanning the gap from very specific to very general systems. In this paper we consider the following common formats: *Applicative* TRSs [24, Section 3.3.5], contain meta-variables, but no bound variables. The prototypical example of an applicative TRS is Curry's combinatory logic. Equipping applicative TRSs with a simple type discipline results in Yamada's *simply typed term rewriting systems* (STTRSs) [28]. Klop's (functional) *combinatory reduction systems* (CRSs) [10], contain both meta-variables and bound variables. The prototypical example of a CRS is Church's λ-calculus. Equipping CRSs with a simple type discipline (and generalizing the notion of substitution), results in Nipkow's *pattern rewrite systems* (PRSs) [13]. We have chosen these formats since they are

1998 ACM Subject Classification: F.4.1, F.4.2.
Key words and phrases: Higher-order rewriting, modularity, termination, normalization.

© C. Appel, V. van Oostrom, and J. G. Simonsen
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.17

relatively well-known and moreover they allow for the 'free' construction of rules, without imposing a priori restrictions on them ensuring termination or confluence.

Modularity in higher-order systems has hitherto only been investigated in isolated cases; Klop proved that confluence is not a modular property in systems that can embed both TRSs and λ-calculus [8], and Klop, van Oostrom and van Raamsdonk showed that acyclicity (a term cannot be reduced to itself) of orthogonal systems, while modular for TRSs, is not a modular property of higher-order systems [9].

In this paper we perform the first systematic study of modularity for higher-order rewriting, see the overview in Table 1. The only non-standard notion employed in the table, is the notion of pattern.

Table 1: Modular properties of first- and higher-order term rewriting system. The results marked (†) are new and proved in this paper.

Property	**TRS**	**STTRS**	**CRS**	**PRS**
Confluence	Yes	No	No	No
Normalization	Yes	No (†)	No (†)	No (†)
Termination	No	No	No	No
Completeness	No	No	No	No
Confluence, for left-linear systems	Yes	Yes	Yes	Yes
Completeness, for left-linear systems	Yes	No (†)	No (†)	No (†)
Unique normal forms	Yes	No (†)	No (†)	No (†)
Normalization, non-duplicating pattern systems	Yes	Yes (†)	?	?
Termination, non-duplicating pattern systems	Yes	Yes (†)	?	No (†)

Definition 1.1. A left-hand side of a rule is a *pattern* if all meta-variables in it are only applied to sequences of pairwise distinct bound variables. A *pattern* rewrite system is one in which all left-hand sides of rules are patterns.

For applicative TRSs and STTRSs the restriction to patterns expresses that meta-variables do not occur *actively* in left-hand sides, i.e. left-hand sides do not have sub-terms of shape $Z\,t$, for Z a meta-variable. Combinatory Logic and all applicative TRSs obtained by Currying are pattern systems. CRSs and PRSs have the condition that left-hand sides of rules be patterns, built into their definition, making matching and unification of left-hand sides first-order like.

The structure of the paper is as follows. We first recapitulate the main positive and negative modularity results from first-order term rewriting, as well as the techniques employed for obtaining them. Next we show by means of a slate of counterexamples, that none of the standard rewriting properties is modular, neither for applicative TRSs, nor for CRSs and PRSs. We end on a positive note, showing that imposing appropriate linearity restriction allows one to regain modularity of some properties, in particular confluence, termination and normalization of STTRSs.

We classify the properties discussed into existence (a normal form can/must/cannot be obtained) and uniqueness (at most one normal form can be obtained) properties. Termination, normalization and acyclicity are existence properties, and confluence and the unique normal forms property are uniqueness properties. Completeness combines both into a unique existence property.

As presenting the counterexamples requires much less technical machinery than the positive results, we postpone the introduction of that machinery to the section containing those positive results. For now, we assume the reader to be familiar with the basic notation for first-order term rewriting systems (TRSs), with simple types, and with the notion of bound variables [3, 24]. This should be sufficient to understand the underlying phenomena, although familiarity with the formats of higher-order term rewriting we treat is an advantage. We refer the reader to [24, Section 3.3.5] for the definition of applicative TRSs, to [28] for STTRSs, to [8, 10, 20] for CRSs, and to [20, 13] for PRSs. Furthermore, we assume the reader to be familiar with the concept of orthogonality in first-order rewriting and its (straightforward) extension to higher-order rewriting; on several occasions we shall use the fact that orthogonal first/higher-order term rewriting systems are confluent [8, 15, 21].

Throughout the paper, we let Σ denote a (first- or higher-order) signature, and $\mathcal{T}$ denote a (first- or higher-order) rewriting system; we equip both of these with integer subscripts when more than one signature or system is needed. We denote by $A \uplus B$ the disjoint union of sets A and B, and we denote by $\mathcal{T}_0 \oplus \mathcal{T}_1 = (\Sigma_0 \uplus \Sigma_1, R_0 \uplus R_1)$ the disjoint union of the rewrite systems $\mathcal{T}_i = (\Sigma_i, R_i)$ for $i \in \{0, 1\}$. A property P of a class $\mathcal{C}$ of rewrite systems is *modular* if $P(\mathcal{T}_0 \oplus \mathcal{T}_1) \;\Leftrightarrow\; P(\mathcal{T}_0)\ \&\ P(\mathcal{T}_1)$ for all $\mathcal{T}_0, \mathcal{T}_1 \in \mathcal{C}$.[1] We employ x, y, z to range over variables for terms of base type, and Z, W, X to range over meta-variables, i.e. variables which yield a term of base type when supplied with sufficiently many terms of the appropriate types. When appropriate, we underline the redex contracted in a rewrite step. Finally, we employ standard rewriting notation as given in [24].

1.1. Modularity in first-order rewriting

The study of modularity in term rewriting was essentially introduced by Toyama in two seminal papers showing, respectively, that confluence is modular for TRSs [26], but that termination is not [25]. Since then, modularity of various properties has been investigated, *e.g.*, normalization (easily seen to be modular, see also [11]), the unique normal forms property (modular [14]), unique normal forms wrt. reduction (not modular [14]), completeness (not modular [25]). For the non-modular properties, restrictions (*e.g.*, left- and/or right-linearity, non-collapsingness) have been put forth that ensure modularity, see for example [22, 27, 12, 23, 16]). Furthermore, modularity has been considered for several varieties of first-order rewriting, and new proofs have been given for modularity of confluence, *cf.*, [19] and its references.

To set the stage for the rest of the paper, we recapitulate Toyama's classical counterexample to modularity of termination for TRSs.

Counterexample 1.2. The single rule TRS

$$f(a, b, x) \to f(x, x, x)$$

is easily proved to be terminating. However, when taking its disjoint union with the, also trivially terminating, two-rule TRS

$$\begin{array}{lcl} g(x, y) & \to & x \\ g(x, y) & \to & y \end{array}$$

[1] An easy consequence of the fact that reduction steps in $\mathcal{T}_i$ can be embedded as reduction steps in $\mathcal{T}_1 \oplus \mathcal{T}_2$, is that each property P studied in this paper holds for $\mathcal{T}_i$ if it holds for $\mathcal{T}_1 \oplus \mathcal{T}_2$. Hence our focus is exclusively on (dis)proving $P(\mathcal{T}_0 \oplus \mathcal{T}_1) \;\Leftarrow\; P(\mathcal{T}_0)\ \&\ P(\mathcal{T}_1)$.

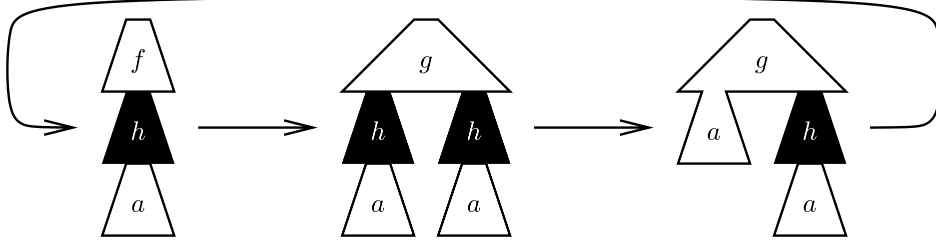


Figure 1: First-order systems: Redex creation can occur by duplicating or collapsing steps. The *rank*, the maximum number of signature changes on a path from the root to a leave in a term, *cannot* increase along reduction. Here, the "white" system is $R_0 = \{f(x) \to g(x,x), g(a,x) \to f(x)\}$ and the "black" system is $R_1 = \{h(y) \to y\}$ (cf. Counterexample 1.2).

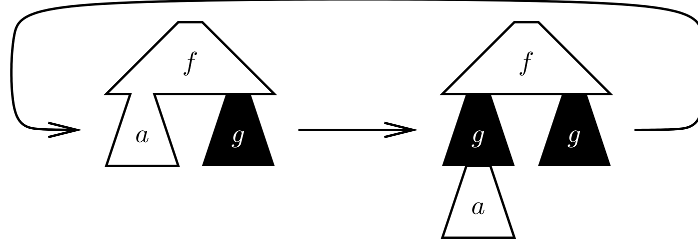


Figure 2: Higher-order systems: Redex creation can *also* occur by application. The rank (measured via the number of signature changes of head symbols of subterms) *may* increase along a reduction. Here, the "white" system is $R_0 = \{f\, a\, Z \to f\, (Z\, a)\, Z\}$ and the "black" system is $R_1 = \{g\, W \to W\}$ (cf. Counterexample 2.5).

we obtain a non-terminating system as witnessed by the cycle:

$$\underline{f(a,b,g(a,b))} \to f(\underline{g(a,b)}, g(a,b), g(a,b)) \to f(a, \underline{g(a,b)}, g(a,b)) \to f(a,b,g(a,b))$$

Intuitively, termination of the first TRS above relies on the *absence* of a term which reduces both to a and b; a property destroyed by the second TRS by its ability to encode non-deterministic choice.

Toyama's counterexample above holds true for any higher-order format embedding TRSs and their rewrite relation, in particular the formats considered in this paper.

The main proof technique for establishing modularity results for first-order TRSs is based on terms in the disjoint union of TRSs being *stratified* in the sense that each term in the disjoint union has a unique decomposition into layers of components residing in either of the TRSs separately, and moreover that this stratification is *preserved* by rewriting in the sense that the *rank*, i.e. the number of layers, cannot increase along a reduction (Figure 1). In the higher-order case, preservation fails due to the presence of rules in which meta-variables can be applied to each other, which allow for *nesting* in the rhs of rules, whence the rank may *increase* along a reduction (Figure 2).

2. Counterexamples to modularity for applicative TRSs

In this section we set the stage for our positive modularity results for applicative TRSs in Sections 5 and 6. This we do by analysing known obstacles for obtaining such results, *cf.* [7], from the perspectives of simply typed STTRS and of pattern rules.

Applicative TRS can be embedded into ordinary (functional) TRSs by viewing the symbols from their signature as nullary function symbols and adjoining one binary function symbol for application. Therefore, one might naïvely expect the modularity results for TRSs to carry over to applicative TRSs. In fact, they do not, the reason being the change in status of the application symbol from being implicit in applicative TRSs to being an explicit element of the signature in their embedding; that is, the embedding of the disjoint union of two applicative TRSs is distinct from the disjoint union of their embeddings [7].

Remark 2.1. To prove, say, confluence of the disjoint union of the confluent applicative TRSs $\{f\,Z \to Z\}$ and $\{g\,W \to W\}$, one might also proceed as follows, *cf.* [24, Section 3.3.5]:
(1) *Uncurrying* yields the confluent functional TRSs $\{f(x) \to x\}$ and $\{g(y) \to y\}$;
(2) Modularity [26] yields confluence of the disjoint union $\{f(x) \to x, g(y) \to y\}$; and
(3) Preservation by *currying* [6] yields confluence of $\{f\,Z \to Z, g\,W \to W\}$, as desired.
The main obstacle following this route is that typically rules of applicative TRSs do contain active (higher-order) variables (this can be seen as the raison d'être of applicative TRSs) and such rules cannot be in the image of the currying transformation.

Uniqueness properties. Unlike what is the case for first-order TRSs, confluence is not a modular property of applicative TRSs as famously shown by Klop [8, Theorem III.1.2.12] who considered the disjoint union of combinatory logic and the applicative TRS $\{D\,Z\,Z \to Z\}$. In order to obtain our positive result of Section 5 we provide some further counterexamples and identify possible causes of non-modularity.

The confluence claims in the counterexamples below are readily verified by standard TRS theory (orthogonality resp. termination and critical pair criteria) applied to the embedding of the applicative TRSs. In our first counterexample the role of combinatory logic in Klop's example is take over by the μ-rule, directly modeling recursion instead of encoding it via a fixed-point combinator in combinatory logic.

Counterexample 2.2. Taking the disjoint union of the confluent applicative TRSs $\{\mu\,Z \to Z\,(\mu\,Z)\}$ and $\{f\,W\,W \to a, f\,W\,(s\,W) \to b\}$ yields a non-confluent system as witnessed by $a \leftarrow f\,(\mu\,s)\,(\mu\,s) \to f\,(\mu\,s)\,(s\,(\mu\,s)) \to b$.

Counterexample 2.3. The disjoint union of the confluent applicative TRSs $\{g\,(Z\,W) \to g\,W\}$ and $\{h\,a \to b\}$ is non-confluent as witnessed by $g\,a \leftarrow g\,(h\,a) \to g\,b$.

Assigning types as $a, b, W : o$, $Z, g, h, s : o \to o$, $\mu : (o \to o) \to o$, and $f : o \to o \to o$ shows that the applicative TRSs in both counterexamples are in fact STTRSs, which entails that confluence is not modular for STTRSs. However, note that the first counterexample employs a non-left-linear rule (*e.g.*, the left-hand side $f\,W\,W$) and the second example a non-pattern rule (with left-hand side $g\,(Z\,W)$). In Section 5 we show that confluence *is* modular for left-linear pattern STTRSs.

The same counterexamples show that the unique normal forms property is not modular for applicative TRSs and STTRSs.

Remark 2.4. The problematic nature of non-pattern rules in applicative TRSs, *i.e.* rules which contain active variables as in the first rule in Counterexample 2.3, is well-known. For instance, adjoining to combinatory logic or the λ-calculus a combinator A defined by $A\,(Z\,W) = Z$, *i.e.* which extracts the function from a function application, immediately renders these calculi inconsistent in the sense that all terms become convertible [4].

Existence properties. Toyama's Counterexample 1.2 to modularity of termination for TRSs carries over immediately to applicative TRSs and even STTRSs, as the terminating TRSs involved can be viewed as terminating STTRSs by assigning appropriate types to the function symbols, *e.g.*, the type $o \to o \to o \to o$ to the ternary function symbol f. But unlike the TRS case also normalization fails for applicative TRSs, under various restrictions, caused by the possibility to apply meta-variables.

In Section 6 we will show that termination and normalization *are* modular for applicative non-duplicating, typable, pattern TRSs. Here we show that any two of them are not sufficient for modularity of termination. The termination claims in the counterexamples below are readily verified by current automated termination tools.

Counterexample 2.5. Taking the disjoint union of the applicative (duplicating) typable pattern TRSs $\{f\,a\,Z \to f\,(Z\,a)\,Z\}$ and $\{g\,W \to W\}$ enables the non-normalizable reduction $f\,a\,g \leftrightarrow f\,(g\,a)\,g$ despite both TRSs being normalizing (even terminating, note that no redex-creation is possible in either of them).

The same example provides a counterexample to modularity of (left-linear, orthogonal) termination, acyclicity, and completeness.

Counterexample 2.6. Taking the disjoint union of the applicative left-and-right-linear (non-typable) pattern TRSs $\{f\,Z\,W \to Z\,a\,f\}$ ('left rotation') and $\{g\,Z\,W \to W\,g\,b\}$ ('right rotation') enables the non-normalizable reduction $f\,g\,b \leftrightarrow g\,a\,f$ despite both TRSs being normalizing (even terminating, based on the insertion of a and b in their 'rotations').

The same example provides a counterexample to modularity of (left-and-right-linear, orthogonal) termination, acyclicity, and completeness.

Counterexample 2.7. Taking the disjoint union of the applicative left-and-right-linear typable (non-pattern) TRSs: $\{f\,(Z\,W) \to Z\,(a\,f)\}$ and $\{g\,(X\,Y) \to Y\,(g\,b)\}$ enables the non-normalizable reduction $f\,(g\,b) \leftrightarrow g\,(a\,f)$ despite both TRSs being normalizing (even terminating, based on the same idea as in the previous counterexample). The TRSs are seen to be typable by assigning types as: $W, b : o$, $Z, f, Y, g : o \to o$, and $a, X : (o \to o) \to o$.

The same example provides a counterexample to modularity of (left-and-right-linear, typable) termination and acyclicity.

3. Counterexamples to modularity for functional CRSs

For PRSs, in general, properties are preserved under signature extensions, i.e. are modular when one of the rewrite systems has no rules at all. The basic idea is to replace each fresh function symbol (from the other signature) by a variable of identical type: If a property does not hold with fresh function symbols, it does not hold with all fresh function symbols replaced by fresh variables, a contradiction. Obviously, this idea fails when the

replacement of function symbols by variables is, for some reason, impossible. In particular, for functional CRSs the replacement is impossible in the absence of meta-variables in terms. More precisely, the rewrite relation of a functional CRS was defined in [10] as a relation on terms *without* meta-variables. As we show in this section, this causes that most properties are not even preserved under signature extensions.

Remark 3.1. The restriction of the rewrite relation to terms without meta-variables is analogous to the restriction of the rewrite relation of first-order TRSs to ground terms. From that perspective, the failure of modularity in case of signature extensions is unsurprising (*e.g.*, normalization is not modular w.r.t. the ground rewrite relation of TRSs; to wit $\{f(a)\to a, f(x) \to f(x)\}$ is ground normalizing but not so when the signature is extended with a constant b). We view our results below in a positive way, as suggesting to change the definition of the rewrite relation of a CRS to include meta-terms, having meta-variables.

Uniqueness properties.

Counterexample 3.2. The CRS given by the rules

$$\begin{array}{rcl} f(f(W)) & \to & f(W) \\ f([x]Z(x)) & \to & f(Z(a)) \\ f([x]Z(x)) & \to & f([x]Z(Z(x))) \end{array}$$

can be shown to be confluent (an easy induction on terms), but is not so after extending the signature with a unary g:

$$f(g(a)) \leftarrow f([x]g(x)) \to f([x]g(g(x))) \to f(g(g(a)))$$

showing non-preservation of confluence.

In effect, the counterexample shows that a CRS can be confluent, i.e. confluent on terms, but not *meta*-confluent, i.e. not confluent on *meta*-terms. This is analogous to the fact that a TRS can be *ground* confluent, but not confluent.

By the same example it follows that the unique normal forms property is not preserved under signature extension either.

Existence properties. For TRSs, termination is preserved under signature extension, as follows by an easy induction on the rank of terms as fresh function symbols partition any term in the disjoint union into terminating components. For PRSs, termination is preserved under signature extension as explained above. For CRSs both of these methods fails, the former because of the lack of an appropriate notion of rank, and the latter because of the absence of fresh meta-variables.

Counterexample 3.3. The CRS having a single, unary function symbol f, and rule

$$f([x][y]Z(x,y)) \quad \to \quad Z([x][y]Z(y,x),[x]x)$$

is terminating, as can be shown by induction on terms (noting that the rewrite relation for CRSs is defined on *terms* not on *meta-terms*, termination of the CRS is shown by an easy induction on terms using that a term may contain *at most one* bound variable, in

the absence of function symbols having arity greater than one). However, extending the signature with a binary symbol g allows to 'swap the roles of x and y'. For instance:

$$\begin{array}{rcl} f([x][y]g(f(x),f(y))) & \to & g(f([x][y]g(f(y),f(x))),f([x]x)) \\ & \to & g(g(f([x]x),f([x][y]g(f(x),f(y)))),f([x]x)) \end{array}$$

The reduction has shape $t \twoheadrightarrow g(g(f([x]x),t),f([x]x))$ for $t = g(g(f([x]x),t),f([x]x))$, giving rise to the *spiralling* reduction:

$$t \twoheadrightarrow g(g(f([x]x),t),f([x]x)) \twoheadrightarrow g(g(f([x]x),g(g(f([x]x),t),f([x]x))),f([x]x)) \twoheadrightarrow \dots$$

showing non-preservation of termination. Note that it is essential for non-termination that $Z(x,y)$ is instantiated by a term containing both x and y, something impossible without function symbols of arity more than 1.

As the CRS is orthogonal and non-erasing, it is terminating iff it is normalizing, whence normalization is not preserved under signature extension either.

Both for TRSs and PRSs, left-linear completeness is preserved under signature extension. For TRSs this is just a special case of modularity of left-linear completeness [27, 23]. For PRSs, it follows by replacing fresh function symbols with fresh variables as explained above. For CRSs, left-linear completeness is not preserved under signature extension: The CRS in Counterexample 3.3 is orthogonal, hence left-linear and confluent, and is terminating, hence complete. However, it is not terminating after adding the fresh symbol g.

4. Counterexamples to modularity for PRSs

In this section we present counterexamples to modularity for Nipkow's pattern rewrite systems.

Since the terms of STTRSs can be embedded directly into PRSs, one might naïvely expect the counterexamples against modularity for STTRSs of Section 2 to carry over to PRSs. In fact, they do not, the reason being the possible presence of abstractions in PRS terms and the ensuing difference in substitution (of higher-order terms).

Example 4.1. As shown in Counterexample 2.5, the system $\{f\,a\,Z \to f\,(Z\,a)\,Z\}$ is terminating when viewed as an STTRS, but not so when viewed as a PRS. To wit, instantiating the meta-variable Z in the rule to $x.x$ yields the infinite looping reduction:

$$f\,a\,(x.x) \to f\,a\,(x.x)$$

Also, PRSs do, unlike CRSs, allow for function variables in terms, hence the counterexamples against modularity for CRSs of Section 3 based on signature extension, do not carry over to PRSs either.

Uniqueness properties. Klop showed in [8] that confluence is not a modular property for CRSs. In particular, his counterexample [8, Theorem III.1.2.10] involves (i) the non-left-linear first-order rule $\{D\,Z\,Z \to Z\}$, and (ii) the β-rule of the λ-calculus.

Similar to what we did in the case of applicative TRSs (Counterexample 2.2), we recast Klop's example as a PRS replacing λ-calculus by the μ-rule, directly modeling recursion instead of encoding it via the fixed-point combinator in the λ-calculus.

Counterexample 4.2. The first-order TRS consisting of the following two rules

$$\begin{array}{rcl} f(x,x) & \to & a \\ f(x,s(x)) & \to & b \end{array}$$

is terminating and has no critical pairs, hence is confluent by Huet's Critical Pair Lemma [5].

However, taking the disjoint union with the orthogonal—hence confluent—single-rule PRS $\mu(x.Z(x)) \to Z(\mu(x.Z(x)))$ yields a non-confluent system as witnessed by:

$$a \leftarrow f(\mu(x.s(x)), \mu(x.s(x))) \to f(\mu(x.s(x)), s(\mu(x.s(x)))) \to b$$

Intuitively, confluence of the TRS above relies both on termination and on the absence of a critical pair involving the two rules, which in turn relies on non-left-linearity and non-convertibility of t and $s(t)$ for any term t. Both of those features are destroyed by the PRS above due to its ability to encode recursion, as witnessed by taking $t = \mu(x.s(x))$.

The unique normal forms property is not modular for PRSs as shown by the same example employed above: As the rewrite systems are confluent they both have the unique normal forms property, but the terms a and b are distinct convertible normal forms in the disjoint union of the TRS and the PRS.

Existence properties. Left-linear completeness is modular for TRSs [27, 23], but fails to be so for PRSs.

Counterexample 4.3. Consider the PRS consisting of the single rule $f(x.x, xy.Z(x,y)) \to g(Z(a, f(x.Z(x,a), xy.Z(x,y))))$ where f and g are second-order symbols and a is a first-order symbol. The PRS is orthogonal, hence confluent. A straightforward analysis of the terms substitutable for Z shows that no redexes can be created (in particular, the sub-term headed by f in the right-hand side cannot give rise to a redex, as that would require $Z(x,y)$ to be instantiated by x which would cause the redex to be 'erased before it is created', so to speak), hence the system is terminating by the Finite Developments Theorem. However, taking the disjoint union with the left-linear and obviously complete TRS consisting of the single rule $h(x,y) \to x$ yields a non-terminating PRS as witnessed by:

$$\underline{f(x.x, xy.h(x,y))} \to g(h(a, f(x.\underline{h(x,a)}, xy.h(x,y)))) \to g(h(a, f(x.x, xy.h(x,y))))$$

Note the reduction sequence above is of the form $t \twoheadrightarrow g(h(a,t))$ for $t = f(x.x, xy.h(x,y))$, hence gives rise to the infinite spiralling reduction:

$$t \twoheadrightarrow g(h(a,t)) \twoheadrightarrow g(h(a, g(h(a,t)))) \twoheadrightarrow g(h(a, g(h(a, g(h(a,t)))))) \twoheadrightarrow \ldots$$

Next we turn our attention to normalization. Normalization is modular in the first-order case as a simple bottom-up argument shows. The result does not extend to PRSs, to wit the following counterexample.

Counterexample 4.4. The PRS consisting of the two rules

$$\begin{array}{rcl} f(x.Z(x), y.y) & \to & f(x.Z(x), y.Z(Z(y))) \\ f(x.x, y.Z(y)) & \to & a \end{array}$$

is normalizing as can be shown by induction on terms substitutable for Z (consider an f-term: if it is not a redex, it cannot become one; if the second rule applies to it, then a is its normal form; if only the first rule applies to it, it can only be applied once). However,

combining it with the trivially normalizing one-rule TRS $g(g(x)) \to x$ yields a system which is not normalizing (it 'reanables' application of the first rule), as witnessed by the cycle:

$$f(x.g(x), y.y) \leftrightarrow f(x.g(x), y.g(g(y)))$$

in which each term is the only possible reduct of the other.

Normalization of the PRS above relies on the left-hand side of its first rule to be non-embeddable into its right-hand side: if it *were* embeddable, the term substituted for $Z(Z(y))$ should be reducible to, and therefore identical, to y, but then the second rule would have been applicable to its lhs as well, ensuring normalization. By adding the projection rule of the TRS above, the left-hand side *can* be embedded, thus destroying normalization.

Counterexample 4.3 witnesses that left-linear completeness is not modular for PRSs.

5. Modularity of confluence in left-linear pattern systems

For first-order left-linear TRSs, modularity of confluence is a trivial consequence of modularity of confluence for arbitrary TRSs. However, since the latter fails in the higher-order case, one may wonder whether left-linearity would suffice to regain modularity of confluence. Indeed it does; the following is a direct corollary of the results of [21].

Theorem 5.1. *Confluence is modular for left-linear pattern systems (applicative TRSs, CRSs, and PRSs).*

The idea of the proof, as presented in the PhD thesis of van Oostrom [17],[2] is to use the Hindley–Rosen Lemma and confluence of each of the PRSs, to reduce confluence of the union to their commutation. The latter holds, because since the signatures are disjoint, and since the rules of the respective PRSs were assumed to be left-linear pattern rules, they are therefore orthogonal *to each other*. The results for applicative pattern TRSs and CRSs follow since these can be embedded faithfully into PRSs.

As a consequence confluence is modular for left-linear pattern STTRSs as well.

Remark 5.2. One may wonder whether confluence is modular for non-duplicating rewrite systems. In the case of CRSs and PRSs the answer is negative [8] (note that the β-rule of the λ-calculus is non-duplicating as a higher-order rule). We leave the question whether confluence is modular for non-duplicating applicative pattern (ST)TRSs to future research.

6. Normalization and termination are modular for non-duplicating pattern STTRSs

In this section we show normalization and termination to be modular for non-duplicating pattern STTRSs. In order to overcome the problem illustrated in Figure 2 that the classical notion of layer will not do as the rank then could increase along reduction, we introduce appropriate notions of *component* and *component-type size*, the idea of the latter being that even though components may become nested (rendering the classical notion of rank useless), this can only be done by means of applying one component to another leading to a decrease in the size of the component types. Since this measure only takes creation

[2] In fact, the more general result is shown there that the (ordinary, non-disjoint) union of two left-linear confluent PRSs is confluent, if the rules are *weakly* orthogonal w.r.t. each other, i.e. all critical pairs are trivial.

of components by means of application into account, not duplication of existing ones, the results are restricted to non-duplicating systems (they have to be in view of Section 2).

In order to stratify mixed applicative terms, we refine the standard notion of a multi-hole context (see *e.g.* [24, Section 2.1.1]) based on classifying symbols into colors. A function symbol belonging to Σ_γ is said to have *color* γ. We will conventionally refer to color 0 as *white*, 1 as *black*, and employ both white ($\square$) and black ($\blacksquare$) typed holes, to be filled by top-white and top-black terms of the appropriate types, respectively. A hole which may be either white or black is denoted by $\boxtimes$. We will view colors both as booleans, applying negation ($\overline{\gamma}$) and exclusive-or ($\gamma_1 \otimes \gamma_2$) to them, and as numbers, multiplying by them.

To illustrate our constructions we make use of the following running example.

Example 6.1. In $\mathcal{T}$ the disjoint unnion of the white rewrite system $\mathcal{T}_0 = \{f\,Z\,W \to Z\,W\}$ with $f:(o\to o)\to o\to o$ and $a:0$, and the black rewrite system $\mathcal{T}_1 = \{g\,a \to a\}$ with $g:o\to o$ and $b:o$, we have the reduction:

$$g\,(f\,(f\,g)\,b) \to g\,(f\,g\,b) \to g\,(g\,b) \to g\,b \to b$$

One can think of components, to be defined next, as the applicative pendant of the notion of layer, well-known from the study of modular properties of first-order term rewriting systems, see *e.g.* [24, Section 5.7.1].

Definition 6.2. For γ either black or white, a γ-*component* is a non-empty context built out of γ-symbols and $\overline{\gamma}$-holes, which does not have active holes, i.e. holes are not applied.

Example 6.3. For the STTRSs of Example 6.1, $f\,\blacksquare\,\blacksquare$ and $f\,(f\,\blacksquare)$ are 0-components, and b, $g\,g$ and $g\,(g\,(g\,\square))$ are 1-components. Non-examples of components are $\blacksquare$ (empty), $f\,g$ (symbols of mixed colors), $\blacksquare\,\blacksquare$ (active hole), $f\,\square$ (same color symbol and hole), and $f\,\blacksquare\,\square$.

We employ C, D, E to range over components. In the following algebraic semantics, we will view every component C of type τ having holes of, from left to right, types $\sigma_1,\ldots,\sigma_n$ as an n-ary function symbol $\boldsymbol{C}:\sigma_1\times\ldots\times\sigma_n\to\tau$ of the *component* signature $\boldsymbol{\Sigma}$. We employ $\boldsymbol{t}$, $\boldsymbol{s}$, $\boldsymbol{u}$ to range over $\boldsymbol{\Sigma}$-terms.

Definition 6.4. The *component* algebra has $\boldsymbol{\Sigma}$-terms as carrier.

$$\begin{array}{rcll} [\![f]\!] & = & f & \\ [\![@]\!](\boldsymbol{C}(\vec{\boldsymbol{t}}),\boldsymbol{D}(\vec{\boldsymbol{s}})) & = & (\boldsymbol{C}\,\boldsymbol{D})(\vec{\boldsymbol{t}},\vec{\boldsymbol{s}}) & \text{if } C,\ D \text{ have the same color} \\ & = & (\boldsymbol{C}\,\boxtimes)(\vec{\boldsymbol{t}},\boldsymbol{D}(\vec{\boldsymbol{s}})) & \text{if } C,\ D \text{ have distinct colors} \end{array}$$

The component algebra gives rise to an obvious bijection mapping each (closed) Σ-term t to its interpretation as $\boldsymbol{\Sigma}$-term, which we indicate by boldface $\boldsymbol{t}$, and vice versa. A term is said to be *top*-white/black if the root symbol of its interpretation is white/black.

Example 6.5. The interpretation of the top-black term $t = g\,(f\,(f\,g)\,b)$ of Example 6.1 is the component term $\boldsymbol{t} = \boldsymbol{C}(\boldsymbol{D}(\boldsymbol{E}_1,\boldsymbol{E}_2))$ with $\boldsymbol{C} = g\,\square$, $\boldsymbol{D} = f\,(f\,\blacksquare)\,\blacksquare$, $\boldsymbol{E}_1 = g$, $\boldsymbol{E}_2 = b$.

Remark 6.6. Our choice to model decomposing terms by means of interpretation into the component algebra is at an abstraction level intermediate between traditional *ad hoc* approaches (involving notions such as *special subterms*, *cf.* [24, Section 5.7.1]), and more recent *categorical* approaches (involving notions such as *monads*, *cf.* [1]), to modularity. It should be interesting to investigate whether the latter apprach, set up to deal with functional TRSs (and collapsing of components), can be adapted to the present setting of applicative TRSs (and application of components).

The idea of the following definition is to measure a term by the 'applicative power' of its components, as expressed by their types. More precisely, terms are measured by pairs the elements of which also take the color of the context in which the term is put, into account: if a top-white (top-black) term is put into a white (black) context, the type of the term itself does not contribute to its measure.

Definition 6.7. The *component-type size* $|t|$[3] of term t is defined to be the pair $|\boldsymbol{t}|$ defined by:

$$|\boldsymbol{C}(\vec{\boldsymbol{t}})| = (\gamma \cdot \#\tau + \#\vec{\boldsymbol{t}}, \overline{\gamma} \cdot \#\tau + \#\vec{\boldsymbol{t}}) \quad \text{if } C : \tau \text{ has color } \gamma$$

where $\#b = 1$, $\#(\sigma \to \tau) = \#\sigma + 1 + \#\tau$, and $\#\boldsymbol{C}(\vec{\boldsymbol{t}}) = \#\tau + \#\vec{\boldsymbol{t}}$ if $C : \tau$. We use $<$ to denote the order induced on $\boldsymbol{\Sigma}$-terms by comparing their component-type sizes by means of the product order of the less-than relation on such pairs of natural numbers. We use $|\,.\,|_0$ ($|\,.\,|_1$) to denote the projection onto the first (second) element of the pair yielded by the component-type size function.

Example 6.8. Since we have $g\,\square : o$, $f\,(f\,\blacksquare)\,\blacksquare : o$, $g : o \to o$, and $b : o$, for the components in Example 6.3, the component-type size $|t|$ of the top-black term $t = g\,(f\,(f\,g)\,b)$ is $(1 \cdot \#o + n, 0 \cdot \#o + n)$ with $n = \#o + \#(o \to o) + \#o$. Therefore $n = 1 + 3 + 1 = 5$ and $|t| = (6, 5)$. That is, only if the (top-black) term t is put into a white context, the type o of the term itself also contributes (1) to the component-type size; otherwise, when put into a black context, only the sub-components contribute (5) to the component-type size.

Lemma 6.9. *The component algebra equipped with $<$ constitutes a well-founded (weakly) monotone Σ-algebra* [24, Definitions 6.2.1,6.4.28].

Proof. Well-foundedness of $<$ is trivial. Application (@) being the only non-nullary symbol it suffices to check its (weak) monotonicity. This follows by calculation. ∎

Example 6.10. Instead of computing directly $|g\,(g\,b)| = (1, 0) < (6, 5) = |g\,(f\,g\,b)|$, the lemma allows to conclude $|g\,(g\,b)| < |g\,(f\,g\,b)|$ from $|g\,b| = (1, 0) < (4, 5) = |f\,g\,b|$ by stricit monotonicity of application in its second argument. The subterm property [24, Definition 6.4.28] does *not* hold: $|f| = (0, 3) \not< (0, 1) = |f\,a|$ (it does for 'special' subterms).

In the traditional terminology of the theory of modularity [16, 24], the following key lemma bounds the component-type size of a term having a *monochrome top*, by the component-type sizes of its *principal/alien subterms*.

Lemma 6.11. *If all symbols in t have color γ and ϕ is a substitution, then for $b \in \{0, 1\}$:*

$$|t^{\phi}|_b \leq (b \otimes \gamma) \cdot \#\tau + \sum_{Z \in t} |\phi(Z)|_{\gamma}$$

with equality holding in case t is a non-empty pattern. Conversely, t is a non-empty pattern in case equality holds and all $\phi(Z)$ are top-$\overline{\gamma}$.[4]

Proof. By induction on t and calculation. ∎

[3]The component-type size is an adaptation of the rank introduced in [2, Definition 8.56].

[4]The summation in the inequality is intended to quantify over *occurrences* of variables in t.

Example 6.12. Let $\phi(Z) = g$ and $\phi(W) = b$.

If $l = f\,Z\,W$ then all symbols in l are white and $|l^\phi| = |f\,g\,b| = (4,5)$. Computing the right-hand side of the inequality in the lemma for, respectively, $b = 0$ and $b = 1$ yields the same pair $(4,5)$. Since the range of ϕ consists of top-white terms, we must conversely have by the lemma that l is a non-empty pattern which indeed it is.

If $r = Z\,W$ them all (none!) symbols in r are white and $|r^\phi| = |g\,b| = (1,0)$. Computing the right-hand side of the inequality in the lemma for, respectively, $b = 0$ and $b = 1$ yields the strictly greater pair $(4,5)$.

Definition 6.13. A rewrite rule is *non-duplicating* if no free (meta-)variable occurs more often in its right-hand side than in its left-hand side.

The following lemma is an analogue of the classical lemma in the theory of modularity of functional TRSs that the *rank* of a term cannot increase along a reduction *cf. e.g.* [26, 24]. In the present applicative case, we have to require rules to be non-duplicating in the light of Counterexample 2.5. The condition entails that rule application can essentially only 'recombine' the components of a term, which will suffice for modularity of termination and normalisation of STTRSs, as for these 'recombination' will entail a decrease in the component-type size.

Lemma 6.14. *If $t \to s$ in the disjoint union of non-duplicating pattern STTRSs, then $|t| \geq |s|$.*

Proof. We claim $|l^\phi| \geq |r^\phi|$, for any rule $l \to r$ with l, r of type τ, and any substitution ϕ. Assuming the claim holds, the result follows by weak monotonicity (Lemma 6.9). The claim itself holds since for $b \in \{0,1\}$

$$|l^\phi|_b = (b \otimes \gamma) \cdot \#\tau + \sum_{Z \in l} |\phi(Z)|_\gamma \geq (b \otimes \gamma) \cdot \#\tau + \sum_{Z \in r} |\phi(Z)|_\gamma \geq |r^\phi|_b \tag{6.1}$$

where the equality holds by Lemma 6.11 using the assumption that l is a pattern and is non-empty (not a single variable) by the general assumption on applicative TRSs, the first inequality holds by the assumption that rules are non-duplicating, and the second inequality holds by Lemma 6.11 again. ∎

Example 6.15. For ϕ, l and r as in Example 6.12 we have $t = l^\phi \to r^\phi = s$ by an application of the rule $l = f\,Z\,W \to Z\,W = r$. As was computed there, indeed $|t| = (4,5) \geq (1,0) = |s|$; the component-size type strictly decreases because the rule combines the arguments substituted for Z and W in its right-hand side $Z\,W$.

We show now that if the component-type size does not decrease across a rewrite step, then the components are not 'combined' and hence the step can be viewed as a step on component symbols. To that end, we define te rewrite relation $\Rightarrow$ on component terms as being generated by the, infintely many, rewrite rules $\boldsymbol{C}(\vec{Z}) \to \boldsymbol{D}(\vec{W})$ for all components C,D of the same color, such that $C[\vec{Z}] \to D[\vec{W}]$.

Lemma 6.16. *If $t \to s$ and $|t| = |s|$ in the disjoint unnion of non-duplicating pattern STTRSs, then $\boldsymbol{t} \Rightarrow \boldsymbol{s}$.*

Proof. We claim $\boldsymbol{t} \Rightarrow \boldsymbol{s}$ if $t = l^\phi$ and $s = r^\phi$, for any rule $l \to r$ with l, r of type τ, and any substitution ϕ such that $|l^\phi| = |r^\phi|$. Assuming the claim holds the result follows by

induction on the derivation of the $\rightarrow$-step as follows. By definition

$$\begin{array}{lcll} |t| & = & (\gamma \cdot \#\tau + \#\vec{\boldsymbol{t}}, \overline{\gamma} \cdot \#\tau + \#\vec{\boldsymbol{t}}) & \text{if } \boldsymbol{t} = \boldsymbol{C}(\vec{\boldsymbol{t}}) \text{ and } C : \tau \text{ has color } \gamma \\ |s| & = & (\delta \cdot \#\tau + \#\vec{\boldsymbol{s}}, \overline{\delta} \cdot \#\tau + \#\vec{\boldsymbol{s}}) & \text{if } \boldsymbol{s} = \boldsymbol{D}(\vec{\boldsymbol{s}}) \text{ and } D : \tau \text{ has color } \delta \end{array}$$

hence $|t| = |s|$ and the fact that $0 \neq \#\tau$ entail $\gamma = \delta$ and $\#\vec{\boldsymbol{t}} = \#\vec{\boldsymbol{s}}$, from which the result easily follows using Definitions 6.4 and 6.7 and the definition of $\Rightarrow$.

It remains to prove the claim. Let ψ and χ be obtained by decomposing the substitution ϕ into γ and $\overline{\gamma}$-components. Formally, ψ and χ such that $\phi = \psi^{\chi}$, are obtained from ϕ by:

$$\begin{array}{lclrcll} \psi(Z) & = & E[\vec{W}_Z] & \chi(W_{Z,i}) & = & t_i & \text{if } \boldsymbol{\phi}(Z) = \boldsymbol{E}(\vec{\boldsymbol{t}}) \text{ and } E \text{ has color } \gamma \\ \psi(Z) & = & W_Z & \chi(Z_Z) & = & \phi(Z) & \text{otherwise} \end{array}$$

Defining $\hat{l} = l^{\psi}$ and $\hat{r} = r^{\psi}$, we have by construction that $\hat{l}$ is a non-empty pattern the symbols of which have color γ, so by the same reasoning as for Equation 6.1, the assumption $|t| = |s|$ yields:

$$|t|_b = |\hat{l}^{\chi}|_b = (b \otimes \gamma) \cdot \#\tau + \sum_{W \in \hat{l}} |\chi(W)|_{\gamma} = (b \otimes \gamma) \cdot \#\tau + \sum_{W \in \hat{r}} |\chi(W)|_{\gamma} = |\hat{r}^{\chi}|_b = |s|_b$$

(From this and the assumption that rules are non-duplicating, it follows that in fact each variable must occur the same number of times in $\hat{l}$ and $\hat{r}$.) From this and Lemma 6.11 it follows that also $\hat{r}$ is a non-empty pattern with symbols of color γ, since by construction r^{ψ} has color γ and all $\chi(W)$ are top-$\overline{\gamma}$. Therefore, to the $\rightarrow$-step $\hat{l} \rightarrow \hat{r}$ the $\Rightarrow$-rule $\boldsymbol{C}(\vec{Z}) \rightarrow \boldsymbol{D}(\vec{W})$ is associated, for the components C and D and vectors of variables $\vec{Z}$ and $\vec{W}$, such that $C[\vec{Z}] = \hat{l}$ and $D[\vec{W}] = \hat{r}$, and the claim follows: $\boldsymbol{t} = \boldsymbol{C}(\boldsymbol{\chi}(\vec{Z})) \Rightarrow \boldsymbol{D}(\boldsymbol{\chi}(\vec{W})) = \boldsymbol{s}$. ∎

Example 6.17. As seen in Examples 6.8 and 6.10 for the step $g\,(f\,(f\,g)\,b) \rightarrow g\,(f\,g\,b)$ it holds $|g\,(f\,(f\,g)\,b)| = (6,5) = |g\,(f\,g\,b)|$. From Example 6.5, the corresponding component terms are $\boldsymbol{t} = \boldsymbol{C}(\boldsymbol{D}(\boldsymbol{E}_1, \boldsymbol{E}_2))$ and $\boldsymbol{s} = \boldsymbol{C}(\boldsymbol{D'}(\boldsymbol{E}_1, \boldsymbol{E}_2))$, using the component symbols given there and $\boldsymbol{D'} = f\,\blacksquare\,\blacksquare$. We indeed have, as per the lemma, $\boldsymbol{t} \Rightarrow \boldsymbol{s}$ by an application of the rule $\boldsymbol{D}(Z,W) \rightarrow \boldsymbol{D}(Z,W)$ obtained from the white step $D[Z,W] = f\,(f\,Z)\,W \rightarrow f\,Z\,W = D[Z,W]$.

Summarizing the above, steps either decrease the component-type size or respect components, in the sense that they can be lifted to the component algebra. This suffices for establishing modularity of termination and normalization for non-duplicating pattern STTRSs.

Theorem 6.18. *Termination is modular for non-duplicating pattern STTRSs.*

Proof. By Lemma 6.14 from some moment on all the terms along an hypothetical infinite $\rightarrow$-reduction must have the same component-type size. We conclude by Lemma 6.16 and the observation that if the rewrite relations $\rightarrow_{\gamma}$ are terminating, then $\Rightarrow$ is seen to be terminating by an application of recursive path orders induced by the precedences induced by $\rightarrow_{\gamma}$ on components. ∎

Theorem 6.19. *Normalisation is modular for non-duplicating pattern STTRSs.*

Proof. By Lemma 6.14 the component-type size cannot increase along $\rightarrow$-reduction, hence any term can be reduced to a term of minimal component-type size, that is, such that any further reduction will leave the component-type size unchanged. We conclude by Lemma 6.16 and the observation that if the rewrite relations $\rightarrow_\gamma$ are normalising, then the corresponding $\Rightarrow$-strategy is seen to be terminating by an application of recursive path orders induced by the precedences induced by the normalising $\rightarrow_\gamma$-strategies on components. ∎

Example 6.20. Each applicative TRS in Example 6.1 is a non-duplicating terminating/normalizing pattern STTRSs. Hence by Theorem 6.18/6.19 so is their disjoint union.

Remark 6.21. Since terms of base type are 'applicatively intert' it might be possible to lift the non-duplicatingness restriction on variables of base type in Theorem 6.19, by appropriately adapting the component-type size. We leave this to future research.

In view of the theorems and of the fact that termination and normalization are modular for non-duplicating functional TRSs [22, 14, 16], one may wonder whether normalization and termination are modular for non-duplicating (or left-and-right-linear) higher-order rewriting systems (CRSs or PRSs). Leaving the other cases to future research (but *cf.* [2, Chapter 9] for some initial and related results), we show modularity of termination fails for left-and-right-linear PRSs because linear PRS rules (such as the β-rule) can still 'embed duplication'.

Counterexample 6.22. The following rules constitute a left-and-right-linear orthogonal PRS:

$$\begin{array}{rcl} f(a,b,Z) & \rightarrow & g(y.f(y,y,y),Z) \\ g(y.Z(y),W) & \rightarrow & Z(W) \end{array}$$

which can be shown terminating by adapting *e.g.* the proof of FD á la Tait of [18], using that by confluence there is no term which can be reduced to both a and b. Note that this non-duplicating PRS can 'simulate' the duplicating first TRS of Counterexample 1.2:

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

hence termination is not preserved when taking the disjoint union with its second TRS.

Acknowledgements. We thank the organisers and participants of the Austria–Japan Summer Workshop on Term Rewriting, 8–13 August 2005, Obergurgl, for the opportunity to present an early version of this paper, and them, Andrzej Filinski, and the anonymous referees for feedback.

References

[1] M. Abbott, N. Ghani, and C. Lüth. Abstract modularity. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*, volume 3467 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2005.

[2] C. Appel. Modularity in higher-order term rewriting. Master's thesis, DIKU, University of Copenhagen, June 2009.

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

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier, 1985.

[5] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, Oct. 1980.

[6] S. Kahrs. Confluence of curried term-rewriting systems. *Journal of Symbolic Computation*, 19(6):601–623, June 1995.

[7] R. Kennaway, J. Klop, R. Sleep, and F.-J. d. Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21(1):15–39, Jan. 1996.

[8] J. Klop. *Combinatory Reduction Systems*. PhD thesis, Utrecht University, 1980.

[9] J. Klop, V. v. Oostrom, and F. v. Raamsdonk. Reduction strategies and acyclicity. In *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, pages 89–112. Springer, 2007.

[10] J. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, Dec. 1993.

[11] M. Kurihara and I. Kaji. Modular term rewriting systems and the termination. *Information Processing Letters*, 34(1):1–4, Feb. 1990.

[12] M. Kurihara and A. Ohuchi. Modularity in noncopying term rewriting. *Theoretical Computer Science*, 152(1):139–169, Dec. 1995.

[13] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, Feb. 1998.

[14] A. Middeldorp. Modular aspects of properties of term rewriting systems related to normal forms. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications (RTA 1989)*, volume 355 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 1989.

[15] T. Nipkow. Higher-order critical pairs. In *Proceedings of the 6th annual IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 342–349, 1991.

[16] E. Ohlebusch. A simple proof of sufficient conditions for the termination of the disjoint union of term rewriting systems. *Bulletin of the European Association for Theoretical Computer Science*, 49:178–183, 1993.

[17] V. v. Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, VU Amsterdam, 1994.

[18] V. v. Oostrom. Take five. Technical Report IR 406, VU Amsterdam, June 1996.

[19] V. v. Oostrom. Modularity of confluence, constructed. In *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR 2008)*, volume 5195 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2008.

[20] V. v. Oostrom and F. v. Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. In *Proceedings of the 1st International Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA 1993)*, volume 814 of *Lecture Notes in Computer Science*, pages 276–304. Springer, 1994.

[21] V. v. Oostrom and F. v. Raamsdonk. Weak orthogonality implies confluence: the higher-order case. In *Proceedings of the 3rd International Symposium on Logical Foundations of Computer Science (LFCS 1994)*, volume 813 of *Lecture Notes in Computer Science*, pages 379–392. Springer, 1994.

[22] M. Rusinowitch. On termination of the direct sum of term-rewriting systems. *Information Processing Letters*, 26(2):65–70, Oct. 1987.

[23] M. Schmidt-Schauss, M. Marchiori, and S. Panitz. Modular termination of r-consistent and left-linear term rewriting systems. *Theoretical Computer Science*, 149(2):361–374, Oct. 1995.

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

[25] Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, May 1987.

[26] Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, Jan. 1987.

[27] Y. Toyama, J. Klop, and H. Barendregt. Termination for the direct sum of left-linear term rewriting systems. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications (RTA 1989)*, volume 355 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 1989.

[28] T. Yamada. Confluence and termination of simply typed term rewriting systems. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *Lecture Notes in Computer Science*, pages 338–352. Springer, 2001.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 33-48
http://rewriting.loria.fr/rta/

CLOSING THE GAP BETWEEN RUNTIME COMPLEXITY AND POLYTIME COMPUTABILITY

MARTIN AVANZINI [1] AND GEORG MOSER [1]

[1] Institute of Computer Science, University of Innsbruck, Austria

E-mail address: {martin.avanzini,georg.moser}@uibk.ac.at

ABSTRACT. In earlier work, we have shown that for confluent term rewrite systems, innermost polynomial runtime complexity induces polytime computability of the functions defined. In this paper, we generalise this result to full rewriting. For that, we again exploit graph rewriting. We give a new proof of the adequacy of graph rewriting for full rewriting that allows for a precise control of the resources copied. In sum we completely describe an implementation of rewriting on a Turing machine. We show that the runtime complexity with respect to rewrite systems is polynomially related to the runtime complexity on a Turing machine.

Our result strengthens the evidence that the complexity of a rewrite system is truthfully represented through the length of derivations. Moreover our result allows the classification of deterministic as well as nondeterministic polytime-computation based on runtime complexity analysis of rewrite systems.

1. Introduction

Recently we see increased interest in studies of the maximal derivation length of term rewrite system, compare for example [9, 10, 15, 11, 14]. We are interested in techniques to automatically classify the complexity of term rewrite systems (TRS for short) and have introduced the *polynomial path order* POP* and extensions of it, cf. [1, 2]. POP* is a restriction of the multiset path order [18] and whenever compatibility of a TRS $\mathcal{R}$ with POP* can be shown then the (innermost) *runtime complexity* of $\mathcal{R}$ is polynomially bounded. Here the runtime complexity of a TRS measures the maximal number of rewrite steps as a function in the size of the initial term, where the initial terms are restricted argument normalised terms (aka *basic* terms).

We have successfully implemented this technique.[1] As a consequence we have a fully automatic (but of course incomplete) procedure that verifies for a given TRS whether it admits at most polynomial runtime complexity. In this paper, we study the question whether

1998 ACM Subject Classification: F 1.2, F 1.3, F 4.2.
Key words and phrases: term rewriting, graph rewriting, complexity analysis, polytime computability.
This research is supported by FWF (Austrian Science Fund) projects P20133-N15.

[1]Our implementation forms part of the *Tyrolean Complexity Tool* (TCT for short). For further information, see http://cl-informatik.uibk.ac.at/software/tct/.

© M. Avanzini and G. Moser
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.33

such techniques are restricted to *runtime complexity*, or whether they can be applied also for the (automated) classification of the *computational complexity* of the functions computed by the given TRS.

For motivation consider the TRS $\mathcal{R}$ given in the next example. It is not difficult to see that $\mathcal{R}$ encodes the function problem FSAT associated to the well-known satisfiability problem SAT. FSAT is complete for the class of *function problems over* NP (FNP for short), compare [16].

Example 1.1. Consider the following TRS $\mathcal{R}$:

1:	$\mathsf{if}(\mathsf{tt}, t, e) \to t$	11:	$\varepsilon = \varepsilon \to \mathsf{tt}$
2:	$\mathsf{if}(\mathsf{ff}, t, e) \to e$	12:	$\mathsf{1}(x) = \mathsf{1}(y) \to x = y$
3:	$\mathsf{choice}(x : xs) \to x$	13:	$\mathsf{1}(x) = \mathsf{0}(y) \to \mathsf{ff}$
4:	$\mathsf{choice}(x : xs) \to \mathsf{choice}(xs)$	14:	$\mathsf{0}(x) = \mathsf{1}(y) \to \mathsf{ff}$
5:	$\mathsf{guess}(\mathsf{nil}) \to \mathsf{nil}$	15:	$\mathsf{0}(x) = \mathsf{0}(y) \to x = y$
6:	$\mathsf{guess}(c : cs) \to \mathsf{choice}(c) : \mathsf{guess}(cs)$	16:	$\mathsf{verify}(\mathsf{nil}) \to \mathsf{tt}$
7:	$\mathsf{in}(x, \mathsf{nil}) \to \mathsf{ff}$	17:	$\mathsf{verify}(l : ls) \to \mathsf{if}(\mathsf{in}(\neg l, ls), \mathsf{ff}, \mathsf{verify}(ls))$
8:	$\mathsf{in}(x, y : ys) \to \mathsf{if}(x = y, \mathsf{tt}, \mathsf{in}(x, ys))$	18:	$\mathsf{sat}'(a) \to \mathsf{if}(\mathsf{verify}(a), a, \mathsf{unsat})$
9:	$\neg \mathsf{1}(x) \to \mathsf{0}(x)$	19:	$\mathsf{sat}(c) \to \mathsf{sat}'(\mathsf{guess}(c))$
10:	$\neg \mathsf{0}(x) \to \mathsf{1}(x)$		

These rules are compatible with POP* and as a result we conclude that the innermost runtime complexity of $\mathcal{R}$ is polynomially bounded.[2]

This leads to the question, whether a characterisation of the runtime complexity of $\mathcal{R}$ suffices to conclude that the functional problem expressed by $\mathcal{R}$ belongs to the class FNP. The purpose of this paper is to provide a positive answer to this question. More precisely, we establish the following results:

1) We re-consider graph rewriting and provide a new proof of the adequacy of graph rewriting for full rewriting. This overcomes obvious inefficiencies of rewriting, when it comes to the duplication of terms.
2) We provide a precise analysis of the resources needed in implementing graph rewriting on a Turing machine (TM for short).
3) Combining these results we show that for a given TRS and for any term t, there exists a normal-form of t that is computable in time $\mathrm{O}(\log(\ell^3) * \ell^7)$ on a deterministic TM. Moreover any normal-form of t is computable in time $\mathrm{O}(\log(\ell^2) * \ell^5)$ on a nondeterministic TM. Here ℓ denotes the maximal length of derivations starting from t.
4) Based on this our main result on the correspondence between polynomial runtime complexity and polytime computability follows.

Our result strengthens the evidence that the complexity of a rewrite system is truthfully represented through the length of derivations. Furthermore, our result allows the classification of deterministic as well as nondeterministic polytime-computation based on runtime complexity analysis of rewrite systems. This extends previous work (see [3]) that

[2]To our best knowledge $\mathsf{T_CT}$ is currently the only complexity tool that can provide a complexity certificate for the TRS $\mathcal{R}$, compare `http://termcomp.uibk.ac.at`.

shows that for confluent TRSs, innermost polynomial runtime complexity induces polytime computability of the functions defined. Moreover, it extends related work by Dal Lago and Martini [8, 7] that studies the complexity of *orthogonal* TRSs, also applying graph rewriting techniques (c.f. also Section 6).

The paper is structured as follows. In Section 2 we present basic notions, in Section 3 we (briefly) recall the central concepts of our employed notion of graph rewriting. The adequacy theorem is provided in Section 4 and in Section 5 we show how rewriting can be implemented efficiently. Finally we obtain our main result in Section 6. Missing proofs are available in the technical report [4].

2. Preliminaries

We assume familiarity with the basics of term rewriting, see [5, 18]. No familiarity with graph rewriting (see [18]) is assumed. Let R be a binary relation on a set S. We write R^+ for the transitive and R^* for the transitive and reflexive closure of R. An element $a \in S$ is *R-minimal* if there exists no $b \in S$ such that $a \mathrel{R} b$. We write $a \mathrel{R^!} b$ if $a \mathrel{R^*} b$ and b is R-minimal.

Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a signature. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted as $\mathcal{T}(\mathcal{F}, \mathcal{V})$ or $\mathcal{T}$ for short. The *size* $|t|$ of a term t is defined as usual. A *term rewrite system* $\mathcal{R}$ over $\mathcal{T}$ is a *finite* set of rewrite rules $l \to r$, such that $l \notin \mathcal{V}$ and $\mathcal{V}\mathrm{ar}(l) \supseteq \mathcal{V}\mathrm{ar}(r)$. We write $\to_{\mathcal{R}}$ for the induced rewrite relation. The set of defined function symbols is denoted as $\mathcal{D}$, while the constructor symbols are collected in $\mathcal{C}$, clearly $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$. We use $\mathrm{NF}(\mathcal{R})$ to denote the set of normal-forms of $\mathcal{R}$. We define the set of *values* $\mathcal{V}\mathrm{al} := \mathcal{T}(\mathcal{C}, \mathcal{V})$, and we define $\mathcal{B} := \{f(v_1, \ldots, v_n) \mid f \in \mathcal{D} \text{ and } v_i \in \mathcal{V}\mathrm{al}\}$ as the set of *basic terms*. Let $\Box$ be a fresh constant. Terms over $\mathcal{F} \cup \{\Box\}$ and $\mathcal{V}$ are called *contexts*. The empty context is denoted as $\Box$. For a context C with n holes, we write $C[t_1, \ldots, t_n]$ for the term obtained by replacing the holes from left to right in C with the terms $t_1, \ldots, t_n$.

A TRS is called *confluent* if for all $s, t_1, t_2 \in \mathcal{T}$ with $s \to_{\mathcal{R}}^* t_1$ and $s \to_{\mathcal{R}}^* t_2$ there exists a term u such that $t_1 \to_{\mathcal{R}}^* u$ and $t_2 \to_{\mathcal{R}}^* u$. The *derivation height* of a terminating term s with respect to a finitely branching relation $\to$ is defined as $\mathrm{dl}(s, \to) := \max\{n \mid \exists t.\ s \to^n t\}$, where $\to^n$ denotes the n-fold application of $\to$. The *runtime complexity function* $\mathrm{rc}_{\mathcal{R}}$ with respect to a TRS $\mathcal{R}$ is defined as $\mathrm{rc}_{\mathcal{R}}(n) := \max\{\mathrm{dl}(t, \to_{\mathcal{R}}) \mid t \in \mathcal{B} \text{ and } |t| \leqslant n\}$.

3. Term Graph Rewriting

In the sequel we introduce the central concepts of *term graph rewriting* or *graph rewriting* for short. We closely follow the presentation of [3], for further motivation of the presented notions we kindly refer the reader to [3]. Let $\mathcal{R}$ be a TRS over a signature $\mathcal{F}$. We keep $\mathcal{R}$ and $\mathcal{F}$ fixed for the remaining of this paper.

A *directed graph* $G = (\mathrm{V}_G, \mathrm{Succ}_G, \mathrm{L}_G)$ over the set $\mathcal{L}$ of *labels* is a structure such that V_G is a finite set, the *nodes* or *vertices*, $\mathrm{Succ}\colon \mathrm{V}_G \to \mathrm{V}_G^*$ is a mapping that associates a node u with an (ordered) sequence of nodes, called the *successors* of u. Note that the sequence of successors of u may be empty: $\mathrm{Succ}_G(u) = []$. Finally $\mathrm{L}_G\colon \mathrm{V}_G \to \mathcal{L}$ is a mapping that associates each node u with its *label* $\mathrm{L}_G(u)$. Typically the set of labels $\mathcal{L}$ is clear from context and not explicitly mentioned. In the following, nodes are denoted by $u, v, \ldots$ possibly followed by subscripts. We drop the reference to the graph G from V_G,

Succ_G, and L_G, i.e., we write $G = (\mathrm{V}, \mathrm{Succ}, \mathrm{L})$ if no confusion can arise from this. Further, we also write $u \in G$ instead of $u \in \mathrm{V}$.

Let $G = (\mathrm{V}, \mathrm{Succ}, \mathrm{L})$ be a graph and let $u \in G$. Consider $\mathrm{Succ}(u) = [u_1, \dots, u_k]$. We call u_i ($1 \leqslant i \leqslant k$) the *i-th successor* of u (denoted as $u \xrightharpoonup{i} u_i$). If $u \xrightharpoonup{i} v$ for some i, then we simply write $u \rightharpoonup v$. A node v is called *reachable* from u if $u \xrightharpoonup{*} v$, where $\xrightharpoonup{*}$ denotes the reflexive and transitive closure of $\rightharpoonup$. We write $\xrightharpoonup{+}$ for $\rightharpoonup \cdot \xrightharpoonup{*}$. A graph G is *acyclic* if $u \xrightharpoonup{+} v$ implies $u \neq v$ and G is *rooted* if there exists a unique node u such that every other node in G is reachable from u. The node u is called the *root* $\mathrm{rt}(G)$ of G. The *size* of G, i.e., the number of nodes, is denoted as $|G|$. The *depth* of G, i.e., the length of the longest path in G, is denoted as $\mathrm{dp}(G)$. We write $G \upharpoonright u$ for the subgraph of G reachable from u.

Let G and H be two term graphs, possibly sharing nodes (see below for the formal definition). We say that G and H are *properly sharing* if $u \in G \cap H$ implies $\mathrm{L}_G(u) = \mathrm{L}_H(u)$ and $\mathrm{Succ}_G(u) = \mathrm{Succ}_H(u)$. If G and H are properly sharing, we write $G \cup H$ for their union.

Definition 3.1. A *term graph* (with respect to $\mathcal{F}$ and $\mathcal{V}$) is an *acyclic* and *rooted* graph $S = (\mathrm{V}, \mathrm{Succ}, \mathrm{L})$ over labels $\mathcal{F} \cup \mathcal{V}$. Let $u \in S$ and suppose $\mathrm{L}(u) = f \in \mathcal{F}$ such that f is k-ary. Then $\mathrm{Succ}(u) = [u_1, \dots, u_k]$. On the other hand, if $\mathrm{L}(u) \in \mathcal{V}$ then $\mathrm{Succ}(u) = []$. We demand that every variable node is *shared*. That is, for $u \in S$ with $\mathrm{L}(u) \in \mathcal{V}$, if $\mathrm{L}(u) = \mathrm{L}(v)$ for some $v \in \mathrm{V}$ then $u = v$.

Below $S, T, \dots$ and L, R, possibly extended by subscripts, always denote term graphs. We write $\mathcal{G}\mathrm{raph}$ for the set of all term graphs with respect to $\mathcal{F}$ and $\mathcal{V}$. Abusing notation from rewriting we set $\mathcal{V}\mathrm{ar}(S) := \{u \mid u \in S, \mathrm{L}(u) \in \mathcal{V}\}$, the set of *variable nodes* in S. We define the term $\mathrm{term}(S)$ *represented* by S as follows: $\mathrm{term}(S) := x$ if $\mathrm{L}(\mathrm{rt}(S)) = x \in \mathcal{V}$ and $\mathrm{term}(S) := f(\mathrm{term}(S \upharpoonright u_1), \dots, \mathrm{term}(S \upharpoonright u_k))$ for $\mathrm{L}(\mathrm{rt}(S)) = f \in \mathcal{F}$ and $\mathrm{Succ}(\mathrm{rt}(S)) = [u_1, \dots, u_k]$. Clearly, we have $|S| \leqslant |\mathrm{term}(S)|$, that is, $|S| = \mathrm{O}(|\mathrm{term}(S)|)$.

We adapt the notion of *positions* in terms to positions in graphs in the obvious way. Positions are denoted as $p, q, \dots$, possibly followed by subscripts. For positions p and q we write pq for their concatenation. We write $p \leqslant q$ if p is a prefix of q, i.e., $q = pp'$ for some position p'. The size $|p|$ of position p is defined as its length. Let $u \in S$ be a node. The set of *positions* $\mathcal{P}os_S(u)$ of u is defined as $\mathcal{P}os_S(u) := \{\varepsilon\}$ if $u = \mathrm{rt}(S)$ and $\mathcal{P}os_S(u) := \{i_1 \cdots i_k \mid \mathrm{rt}(S) \xrightharpoonup{i_1} \cdots \xrightharpoonup{i_k} u\}$ otherwise. The set of all positions in S is $\mathcal{P}os_S := \bigcup_{u \in S} \mathcal{P}os_S(u)$. Note that $\mathcal{P}os_S$ coincides with the set of positions of $\mathrm{term}(S)$. If $p \in \mathcal{P}os_S(u)$ we say that u *corresponds* to p. In this case we also write $S \upharpoonright p$ for the subgraph $S \upharpoonright u$. This is well defined since exactly one node corresponds to a position p. One easily verifies $\mathrm{term}(S \upharpoonright p) = \mathrm{term}(S)|_p$ for all $p \in \mathcal{P}os_S$. We say that u is (strictly) *above* a position p if u corresponds to a position q with $q \leqslant p$ ($q < p$). Conversely, the node u is *below* p if u corresponds to q with $p \leqslant q$.

By exploiting different degrees of *sharing*, a term t can often be represented by more than one term graph. Let S be a term graph and let $u \in S$ be a node. We say that u is *shared* if the set of positions $\mathcal{P}os_S(u)$ is not singleton. Note that in this case, the node u represents more than one subterm of $\mathrm{term}(S)$. If $\mathcal{P}os_S(u)$ is singleton, then u is *unshared*.

The node u is *minimally shared* if it is either unshared or a variable node (recall that variable nodes are always shared). We say u is *maximally shared* if $\mathrm{term}(S \upharpoonright u) = \mathrm{term}(S \upharpoonright v)$ implies $u = v$ for all nodes $v \in S$. The term graph S is called *minimally sharing* (*maximally sharing*) if all nodes $u \in S$ are minimally shared (maximally shared). Let s be a term. We collect all minimally sharing term graphs representing s in the set $\triangle(s)$. Maximally sharing

term graphs representing s are collected in $\nabla(s)$. Observe that for $S \in \triangle(s)$, we have $|s| = \mathrm{O}(|S|)$.

We now introduce a notion for replacing a subgraph $S \upharpoonright u$ of S by a graph H.

Definition 3.2. Let S be a term graph and let $u, v \in S$ be two nodes. Then $S[u \leftarrow v]$ denotes the *redirection* of node u to v: define the mapping r such that $r(u) := v$ and $r(w) := w$ for all $w \in S \backslash \{u\}$. Set $\mathrm{V}' := (\mathrm{V}_S \cup \{v\}) \backslash \{u\}$ and for all $w \in \mathrm{V}'$, $\mathrm{Succ}'(w) := r^*(\mathrm{Succ}_S(w))$ where r^* is the extension of r to sequences. Finally, set $S[u \leftarrow v] := (\mathrm{V}', \mathrm{Succ}', \mathrm{L}_S)$.

Let H be a rooted graph over $\mathcal{F} \cup \mathcal{V}$. We define $S[H]_u := (S[u \leftarrow \mathrm{rt}(H)] \cup H) \upharpoonright v$ where $v = \mathrm{rt}(H)$ if $u = \mathrm{rt}(S)$ and $v = \mathrm{rt}(S)$ otherwise. Note that $S[H]_u$ is again a term graph if $u \notin H$ and H acyclic.

The following notion of *term graph morphism* plays the role of substitutions.

Definition 3.3. Let L and S be two term graphs. A *morphism* from L to S (denoted $m\colon L \to S$) is a function $m\colon \mathrm{V}_L \to \mathrm{V}_S$ such that $m(\mathrm{rt}(L)) = \mathrm{rt}(S)$, and for all $u \in L$ with $\mathrm{L}_L(u) \in \mathcal{F}$, (i) $\mathrm{L}_L(u) = \mathrm{L}_S(m(u))$ and (ii) $m^*(\mathrm{Succ}_L(u)) = \mathrm{Succ}_S(m(u))$.

The next lemma follows from Definition 3.3.

Lemma 3.4. *If $m\colon L \to S$ then for any $u \in L$ we have $m\colon L \upharpoonright u \to S \upharpoonright m(u)$.*

Let $m\colon L \to S$ be a morphism from L to S. The *induced substitution* $\sigma_m\colon \mathcal{V}\mathrm{ar}(L) \to \mathcal{T}$ is defined as $\sigma_m(x) := \mathrm{term}(S \upharpoonright m(u))$ for any $u \in L$ such that $\mathrm{L}(u) = x \in \mathcal{V}$. As an easy consequence of Lemma 3.4 we obtain the following.

Lemma 3.5. *Let L and S be term graphs, and suppose $m\colon L \to S$ for some morphism m. Let σ_m be the substitution induced by m. Then $\mathrm{term}(L)\sigma_m = \mathrm{term}(S)$.*

Proof. The lemma has been shown in [3, Lemma 14]. ∎

We write $S \geqslant_m T$ (or $S \geqslant T$ for short) if $m\colon S \to T$ is a morphism such that for *all* $u \in \mathrm{V}_S$, Property (i) and Property (ii) in Definition 3.3 are fulfilled. For this case, S and T represent the same term. We write $S >_m T$ (or $S > T$ for short) when the graph morphism m is additionally *non-injective*. If both $S \geqslant T$ and $T \geqslant S$ holds then S and T are *isomorphic*, in notation $S \cong T$. Recall that $|S|$ denotes the number of nodes in S.

Lemma 3.6. *For all term graph S and T, $S \geqslant_m T$ implies $\mathrm{term}(S) = \mathrm{term}(T)$ and $|S| \geqslant |T|$. If further $S >_m T$ holds then $|S| > |T|$.*

Let L and R be two properly sharing term graphs. Suppose $\mathrm{rt}(L) \notin \mathcal{V}\mathrm{ar}(L)$, $\mathcal{V}\mathrm{ar}(R) \subseteq \mathcal{V}\mathrm{ar}(L)$ and $\mathrm{rt}(L) \notin R$. Then the graph $L \cup R$ is called a *graph rewrite rule* (*rule* for short), denoted by $L \to R$. The graph L, R denotes the left-hand, right-hand side of $L \to R$ respectively. A *graph rewrite system* (*GRS* for short) $\mathcal{G}$ is a set of graph rewrite rules.

Let $\mathcal{G}$ be a GRS, let $S \in \mathcal{G}\mathrm{raph}$ and let $L \to R$ be a rule. A rule $L' \to R'$ is called a *renaming* of $L \to R$ with respect to S if $(L' \to R') \cong (L \to R)$ and $\mathrm{V}_S \cap \mathrm{V}_{L' \to R'} = \varnothing$. Let $L' \to R'$ be a renaming of a rule $(L \to R) \in \mathcal{G}$ for S, and let $u \in S$ be a node. We say S *rewrites* to T at *redex* u with rule $L \to R$, denoted as $S \longrightarrow_{\mathcal{G},u,L\to R} T$, if there exists a morphism $m\colon L' \to S \upharpoonright u$ and $T = S[m(R')]_u$. Here $m(R')$ denotes the structure obtained by replacing in R' every node $v \in \mathrm{dom}(m)$ by $m(v) \in S$, where the labels of $m(v) \in m(R')$ are the labels of $m(v) \in S$. We also write $S \longrightarrow_{\mathcal{G},p,L\to R} T$ if $S \longrightarrow_{\mathcal{G},u,L\to R} T$ for position p corresponding to u in S. We set $S \longrightarrow_{\mathcal{G}} T$ if $S \longrightarrow_{\mathcal{G},u,L\to R} T$ for some $u \in S$ and

$(L \to R) \in \mathcal{G}$. The relation $\longrightarrow_{\mathcal{G}}$ is called the *graph rewrite relation* induced by $\mathcal{G}$. Again abusing notation, we denote the set of normal-forms with respect to $\longrightarrow_{\mathcal{G}}$ as $\mathrm{NF}(\mathcal{G})$.

4. Adequacy of Graph Rewriting for Term Rewriting

In earlier work [3] we have shown that graph rewriting is adequate for innermost rewriting without further restrictions on the studied TRS $\mathcal{R}$. In this section we generalise this result to full rewriting. The adequacy theorem presented here (see Theorem 4.15) is not essentially new. Related results can be found in the extensive literature, see for example [18]. In particular, in [17] the adequacy theorem is stated for full rewriting and unrestricted TRSs. In this work, we take a fresh look from a complexity related point of view.

We give a new proof of the adequacy of graph rewriting for full rewriting that allows for a precise control of the resources copied. This is essential for the accurate characterisation of the implementation of graph rewriting given in Section 5.

Definition 4.1. The *simulating graph rewrite system* $\mathcal{G}(\mathcal{R})$ of $\mathcal{R}$ contains for each rule $(l \to r) \in \mathcal{R}$ some rule $L \to R$ such that $L \in \triangle(l)$, $R \in \triangle(r)$ and $\mathrm{V}_L \cap \mathrm{V}_R = \mathcal{V}\mathrm{ar}(R)$.

The next two lemmas establish soundness in the sense that derivations with respect to $\mathcal{G}(\mathcal{R})$ correspond to $\mathcal{R}$-derivations.

Lemma 4.2. *Let S be a term graph and let $L \to R$ be a renaming of a graph rewrite rule for S, i.e., $S \cap R = \varnothing$. Suppose $m\colon L \to S$ for some morphism m and let σ_m be the substitution induced by m. Then* $\mathrm{term}(R)\sigma_m = \mathrm{term}(T)$ *where* $T := (m(R) \cup S)\upharpoonright \mathrm{rt}(m(R))$.

Proof. The lemma has been shown in [3, Lemma 15]. ∎

In Section 2 we introduced $\Box$ as designation of the empty context. Below we write $\Box$ for the unique (up-to isomorphism) graph representing the constant $\Box$.

Lemma 4.3. *Let S and T be two properly sharing term graphs, let $u \in S \setminus T$ and $C =$* $\mathrm{term}(S[\Box]_u)$. *Then* $\mathrm{term}(S[T]_u) = C[\mathrm{term}(T), \dots, \mathrm{term}(T)]$.

Proof. The lemma has been shown in [3, Lemma 16]. Note that the set of positions of $\Box$ in C corresponds to $\mathcal{P}os_S(u)$. ∎

For non-left-linear TRSs $\mathcal{R}$, $\longrightarrow_{\mathcal{G}(\mathcal{R})}$ does not suffice to mimic $\to_{\mathcal{R}}$. This is clarified in the following example.

Example 4.4. Consider the TRS $\mathcal{R} := \{\mathsf{f}(x) \to \mathsf{eq}(x, \mathsf{a});\ \mathsf{eq}(x, x) \to \top\}$. Then $\mathcal{R}$ admits the derivation

$$\mathsf{f}(\mathsf{a}) \to_{\mathcal{R}} \mathsf{eq}(\mathsf{a}, \mathsf{a}) \to_{\mathcal{R}} \top$$

but $\mathcal{G}(\mathcal{R})$ cannot completely simulate the above sequence:

```
f                 eq
|  --->G(R)      /  \     ∈ NF(G(R))
a               a    a
```

Let $L \to R$ be the rule in $\mathcal{G}(\mathcal{R})$ corresponding to $\mathsf{eq}(x, x) \to \top$, and let S, $\mathrm{term}(S) = \mathsf{eq}(\mathsf{a}, \mathsf{a})$, be the second graph in the above sequence. Then $L \to R$ is inapplicable as we cannot simultaneously map the unique variable node in L to both leaves in S via a graph morphism. Note that the situation can be repaired by sharing the two arguments in S.

For maximally sharing graphs S we can prove that redexes of $\mathcal{R}$ and (positions corresponding to) redexes of $\mathcal{G}(\mathcal{R})$ coincide. This is a consequence of the following lemma.

Lemma 4.5. *Let l be a term and $s = l\sigma$ for some substitution σ. If $L \in \triangle(l)$ and $S \in \nabla(s)$, then there exists a morphism $m\colon L \to S$. Further, $\sigma(x) = \sigma_m(x)$ for the induced substitution σ_m and all variables $x \in \mathcal{V}\mathrm{ar}(l)$.*

Proof. We prove the lemma by induction on l. It suffices to consider the induction step. Let $l = f(l_1, \ldots, l_k)$ and $s = f(l_1\sigma, \ldots, l_k\sigma)$. Suppose $\mathrm{Succ}_L(\mathrm{rt}(L)) = [u_1, \ldots, u_k]$ and $\mathrm{Succ}_S(\mathrm{rt}(S)) = [v_1, \ldots, v_k]$. By induction hypothesis there exist morphisms $m_i\colon L \upharpoonright u_i \to S \upharpoonright v_i$ ($1 \leqslant i \leqslant k$) of the required form. Define $m\colon \mathrm{V}_L \to \mathrm{V}_S$ as follows. Set $m(\mathrm{rt}(L)) = \mathrm{rt}(S)$ and for $w \neq \mathrm{rt}(L)$ define $m(w) = m_i(w)$ if $w \in \mathrm{dom}(m_i)$. We claim $w \in (\mathrm{dom}(m_i) \cap \mathrm{dom}(m_j))$ implies $m_i(w) = m_j(w)$. For this, suppose $w \in (\mathrm{dom}(m_i) \cap \mathrm{dom}(m_j))$. Since $L \in \triangle(l)$, only variable nodes are shared, hence w needs to be a variable node, say $\mathrm{L}_L(w) = x \in \mathcal{V}$. Then

$$\mathrm{term}(S \upharpoonright m_i(w)) = \sigma_{m_i}(x) = \sigma(x) = \sigma_{m_j}(x) = \mathrm{term}(S \upharpoonright m_j(w))$$

by definition and induction hypothesis. As $S \in \nabla(s)$ is maximally shared, $m_i(w) = m_j(w)$ follows. We conclude m is a well-defined morphism, further $m\colon L \to S$. ∎

A second problem is introduced by non-eager evaluation. Consider the following.

Example 4.6. Let $\mathcal{R} := \{\mathsf{dup}(x) \to \mathsf{c}(x, x);\ \mathsf{a} \to \mathsf{b}\}$. Then $\mathcal{R}$ admits the derivation

$$\mathsf{dup}(\mathsf{a}) \to_{\mathcal{R}} \mathsf{c}(\mathsf{a}, \mathsf{a}) \to_{\mathcal{R}} \mathsf{c}(\mathsf{b}, \mathsf{a})$$

but applying the corresponding rules in $\mathcal{G}(\mathcal{R})$ yields:

```
dup               c               c
 |   ⟶G(R)       ( )    ⟶G(R)    ( )
 a                a               b
```

Application of the first rule produces a shared redex. Consequently the second step amounts to a parallel step in $\mathcal{R}$.

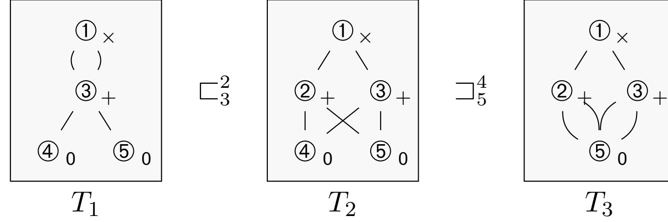
To prove adequacy of graph rewriting for term rewriting and unrestricted TRSs, we follow the standard approach [18, 17] where *folding* (also called *collapsing*) and *unfolding* (also referred to as *copying*) is directly incorporated in the graph rewrite relation. Unlike in the cited literature, we employ a very restrictive form of folding and unfolding. To this extend, we define for positions p relations $\blacktriangleright_p$ and $\lhd_p$ on term graphs. Both relations preserve term structure. However, when $S \blacktriangleright_p T$ holds then the subgraph $T \upharpoonright p$ admits strictly more sharing than $S \upharpoonright p$. Conversely, when $S \lhd_p T$ holds, nodes above p in T admit less sharing than nodes above p in S. Extending the graph rewrite relation $\longrightarrow_{\mathcal{G}(\mathcal{R}),p}$ by $\blacktriangleright_p$ and $\lhd_p$ addresses both problems highlighted in Example 4.4 and Example 4.6.

The relations $\blacktriangleright_p$ and $\lhd_p$ are based on *single step* approximations $\sqsupseteq_v^u$ of $>_m$.

Definition 4.7. Let $\succ$ denote some total quasi-order on nodes, let $\succcurlyeq$ denote the reflexive closure of $\succ$. Let S be a term graph, and let $u, v \in S$ be nodes satisfying $u \succcurlyeq v$. We define $S \sqsupseteq_v^u T$ for term graph T if $S \geqslant_m T$ for the morphism m identifying u and v, more precisely, $m(u) = v$ and $m(w) = w$ for all $w \in S \setminus \{u\}$. We define $S \sqsupset_v^u T$ if $S \sqsupseteq_v^u T$ and $u \neq v$.

We write $S \sqsupseteq_v T$ ($S \sqsupset_v T$) if there exists $u \in S$ such that $S \sqsupseteq_v^u T$ ($S \sqsupset_v^u T$) holds. Similar $S \sqsupseteq T$ ($S \sqsupset T$) if there exist nodes $u, v \in S$ such that $S \sqsupseteq_v^u T$ ($S \sqsupset_v^u T$) holds.

Example 4.8. Consider the term $t = (0 + 0) \times (0 + 0)$. Then t is represented by the following three graphs that are related by $\sqsubset_3^2$ and $\sqsupset_5^4$ respectively.



That is, the term graph T_2 is obtained from T_1 by copying node 3, introducing the fresh node 2. The graph T_3 is obtained from T_2 by collapsing node 4 onto node 5.

Suppose $S \sqsupset_v^u T$. Then the morphism underlying $\sqsupset_v^u$ amounts to the identity on $\mathrm{V}_S \setminus \{u\}$. In particular, it defines the identity on successors of $u, v \in S$. Thus the following is immediate.

Lemma 4.9. *Let S be a term graph, and let $u, v \in S$ be two distinct nodes. Then there exists a term graph T such that $S \sqsupset_v^u T$ if and only if $\mathrm{L}_S(u) = \mathrm{L}_S(v)$ and $\mathrm{Succ}_S(u) = \mathrm{Succ}_S(v)$.*

The restriction $u \not\geqslant v$ was put onto $\sqsupset_v^u$ so that $\sqsupseteq_v$ enjoys the following diamond property. Otherwise, the peak $\sqsubset_v^u \cdot \sqsupset_u^v \subseteq \cong$ cannot be joined.

Lemma 4.10. $\sqsubseteq_u \cdot \sqsupseteq_v \;\subseteq\; \sqsupseteq_{w_1} \cdot \sqsubseteq_{w_2}$ *where $w_1, w_2 \in \{u, v\}$.*

Proof. Assume $T_1 \sqsubseteq_u^{u'} S \sqsupseteq_v^{v'} T_2$ for some term graphs S, T_1 and T_2. The only non-trivial case is $T_1 \sqsubset_u^{u'} S \sqsupset_v^{v'} T_2$ for $u' \neq v'$ and $u \neq v$. We prove $T_1 \sqsupset_{w_1} \cdot \sqsubset_{w_2} T_2$ for $w_1, w_2 \in \{u, v\}$ by case analysis. We highlight two interesting cases. The remaining cases follow by similar reasoning, c.f. [4].

- Case $T_1 \sqsubset_w^{u'} S \sqsupset_w^{v'} T_2$ for $v' \neq u'$. We claim $T_1 \sqsupset_w^{v'} \cdot \sqsubset_w^{u'} T_2$. Let m_1 be the morphism underlying $\sqsubset_w^{u'}$ and let m_2 be the morphism underlying $\sqsupset_w^{v'}$ (c.f. Definition 4.7). We first show $\mathrm{L}_{T_1}(v') = \mathrm{L}_{T_1}(w)$ and $\mathrm{Succ}_{T_1}(v') = \mathrm{Succ}_{T_1}(w)$. Using Lemma 4.9, $S \sqsupset_w^{v'} T_2$ yields $\mathrm{L}_S(v') = \mathrm{L}_S(w)$. Employing $v' \neq u'$ and $w \neq u'$ we see
$$\begin{aligned}\mathrm{L}_{T_1}(v') = \mathrm{L}_{T_1}(m_1(v')) = \mathrm{L}_S(v')& \\ = \mathrm{L}_S(w) = \mathrm{L}_{T_1}(m_1(w)) = \mathrm{L}_{T_1}(w)\ .\end{aligned}$$
where we employ $m_1(v') = v'$ and $m_1(w) = w$. Again by Lemma 4.9, we see $\mathrm{Succ}_S(u') = \mathrm{Succ}_S(w)$ and $\mathrm{Succ}_S(v') = \mathrm{Succ}_S(w)$ by the assumption $T_1 \sqsubset_w^{u'} S \sqsupset_w^{v'} T_2$. We conclude $\mathrm{Succ}_S(v') = \mathrm{Succ}_S(w)$ and thus
$$\begin{aligned}\mathrm{Succ}_{T_1}(v') = \mathrm{Succ}_{T_1}(m_1(v')) = m_1^*(\mathrm{Succ}_S(v'))& \\ = m_1^*(\mathrm{Succ}_S(w)) = \mathrm{Succ}_{T_1}(m_1(w)) = \mathrm{Succ}_{T_1}(w)\ .\end{aligned}$$
By Lemma 4.9 we obtain term graph U_1 such that $T_1 \sqsupset_w^{v'} U_1$. Symmetrically, we can prove $T_2 \sqsupset_w^{u'} U_2$ for some term graph U_2. Hence $T_1 \sqsupset_w^{v'} \cdot \sqsubset_w^{u'} T_2$ holds if $U_1 = U_2$. To prove the latter, one shows $m_2 \cdot m_1 = m_1 \cdot m_2$ by a straightforward case analysis.
- Case $T_1 \sqsubset_u^{u'} S \sqsupset_v^{v'} T_2$ for pairwise distinct u', u, v' and v. We show $T_1 \sqsupset_v^{v'} \cdot \sqsubset_u^{u'} T_2$. Let m be the morphism underlying $\sqsupset_u^{u'}$. Observe $m(v) = v$ and $m(v') = v'$ by our assumption. Hence $\mathrm{L}_{T_1}(v') = \mathrm{L}_S(v') = \mathrm{L}_S(v) = \mathrm{L}_{T_1}(v)$ and $\mathrm{L}_{T_1}(v') = m^*(\mathrm{L}_S(v')) = m^*(\mathrm{L}_S(v)) = \mathrm{L}_{T_1}(v)$. We obtain $T_1 \sqsupset_v^{v'} U_1$ and symmetrically $T_2 \sqsupset_u^{u'} U_2$ for some term graphs U_1 and U_2. Finally, one verifies $U_1 = U_2$ by case analysis as above. ∎

The above lemma implies confluence of $\sqsupseteq$. Since $\sqsupset^* = \sqsupseteq^*$, $\sqsupset$ is also confluent.

Definition 4.11. Let S be a term graph and let p be a position in S. We say that S *folds strictly below* p *to the term graph* T, in notation $S \blacktriangleright_p T$, if $S \sqsupset_v^u T$ for nodes $u, v \in S$ strictly below p in S. The graph S *unfolds above* p *to the term graph* T, in notation $S \lhd_p T$, if $S \sqsubset_v^u T$ for some unshared node $u \in T$ above p, i.e., $\mathcal{P}os_T(u) = \{q\}$ for $q \leqslant p$.

Example 4.12. Reconsider the term graphs T_1, T_2 and T_3 with $T_1 \sqsubset_3^2 T_2 \sqsupset_5^4 T_3$ from Example 4.8. Then $T_1 \lhd_2 T_2$ since node 3 is an unshared node above position 2 in T_2. Further $T_2 \blacktriangleright_2 T_3$ since both nodes 4 and 5 are strictly below position 2 in T_2.

Note that for $S \sqsupset_v^u T$ the sets of positions $\mathcal{P}os_S$ and $\mathcal{P}os_T$ coincide, thus the n-fold composition $\lhd_p^n$ of $\lhd_p$ (and the n-fold composition $\blacktriangleright_p^n$ of $\blacktriangleright_p$) is well-defined for $p \in \mathcal{P}os_S$. In the next two lemmas we prove that relations $\lhd_p$ and $\blacktriangleright_p$ fulfil their intended purpose.

Lemma 4.13. *Let S be a term graph and p a position in S. If S is $\lhd_p$-minimal then the node corresponding to p is unshared.*

Proof. By way of contradiction, suppose S is $\lhd_p$-minimal but the node w corresponding to p is shared. We construct T such that $S \lhd_p T$. We pick an unshared node $v \in S$, and shared node $v_i \in S$, above p such that $v \rightharpoonup v_i$. By a straightforward induction on p we see that v and v_i exist as w is shared. For this, note that at least the root of S is unshared.

Define $T := (\mathrm{V}_T, \mathrm{L}_T, \mathrm{Succ}_T)$ as follows: let u be a fresh node such that $u \succ v_i$. set $\mathrm{V}_T := \mathrm{V}_S \cup \{u\}$; set $\mathrm{L}_T(u) := \mathrm{L}_S(v_i)$ and $\mathrm{Succ}_T(u) := \mathrm{Succ}_S(v_i)$; further replace the edge $v \xrightharpoonup{i} v_i$ by $v \xrightharpoonup{i} u$, that is, set $\mathrm{L}_T(v) := [v_1, \ldots, u, \ldots, v_l]$ for $\mathrm{L}_S(v) = [v_1, \ldots, v_i, \ldots, v_l]$. For the remaining cases, define $\mathrm{L}_T(w) := \mathrm{L}_S(u)$ and $\mathrm{Succ}_T(w) := \mathrm{Succ}_S(w)$. One easily verifies $T \sqsupset_{v_i}^u S$. Since by way of construction u is an unshared node above p, $S \lhd_p T$ holds. ∎

Lemma 4.14. *Let S be a term graph, let p be a position in S. If S is $\blacktriangleright_p$-minimal then the subgraph $S \restriction p$ is maximally shared.*

Proof. Suppose $S \restriction p$ is not maximally shared. We show that S is not $\blacktriangleright_p$-minimal. Pick some node $u \in S \restriction p$ such that there exists a distinct node $v \in S \restriction p$ with $\mathrm{term}(S \restriction u) = \mathrm{term}(S \restriction v)$. For that we assume that u is $\rightharpoonup$*-minimal* in the sense that there is no node u' with $u \xrightharpoonup{+} u'$ such that u' would fulfil the above property. Clearly $\mathrm{L}_S(u) = \mathrm{L}_S(v)$ follows from $\mathrm{term}(S \restriction u) = \mathrm{term}(S \restriction v)$. Next, suppose $u \xrightharpoonup{i} u_i$ and $v \xrightharpoonup{i} v_i$ for some nodes $u_i \neq v_i$. But then u_i contradicts minimality of u, and so we conclude $u_i = v_i$. Consequently $\mathrm{Succ}_S(u) = \mathrm{Succ}_S(v)$ follows as desired. Without loss of generality, suppose $u \succ v$. By Lemma 4.9 there exists a term graph T such that $S \sqsupset_v^u T$. Since $u, v \in S \restriction p$, $S \blacktriangleright_p T$ follows. ∎

Theorem 4.15 (Adequacy)**.** *Let s be a term and let S be a term graph such that* $\mathrm{term}(S) = s$. *Then*

$$s \to_{\mathcal{R},p} t \text{ if and only if } S \lhd_p^! \cdot \blacktriangleright_p^! \cdot \longrightarrow_{\mathcal{G}(\mathcal{R}),p} T$$

for some term graph T with $\mathrm{term}(T) = t$.

Proof. First, we consider the direction from right to left. Suppose $S \lhd_p^! U \blacktriangleright_p^! V \longrightarrow_{\mathcal{G}(\mathcal{R}),p} T$. Note that $\blacktriangleright_p$ preserves $\lhd_p$-minimality. We conclude V is $\lhd_p$-minimal as U is. Let $v \in V$ be the node corresponding to p. By Lemma 4.13 we see $\mathcal{P}os_U(v) = \{p\}$. Now consider the step $V \longrightarrow_{\mathcal{G}(\mathcal{R}),p} T$. There exists a renaming $L' \to R'$ of $(L \to R) \in \mathcal{G}(\mathcal{R})$ such that $m \colon L' \to V \restriction v$ is a morphism and $T = V[m(R')]_v$. Set $l := \mathrm{term}(L')$ and $r := \mathrm{term}(R')$, by

definition $(l \to r) \in \mathcal{R}$. By Lemma 3.5 we obtain $l\sigma_m = \mathrm{term}(V \restriction v)$ for the substitution σ_m induced by the morphism m. Define the context $C := \mathrm{term}(V[\Box]_v)$. As v is unshared, C admits exactly one occurrence of $\Box$, moreover the position of $\Box$ in C is p. By Lemma 4.3,

$$\mathrm{term}(V) = \mathrm{term}(V[V \restriction v]_v) = C[\mathrm{term}(V \restriction v)] = C[l\sigma_m] .$$

Set $T_v := (m(R') \cup V) \restriction \mathrm{rt}(m(R'))$, and observe $T = V[m(R')]_v = V[T_v]_v$. Using Lemma 4.3 and Lemma 4.2 we see

$$\mathrm{term}(T) = \mathrm{term}(V[T_v]_v) = C[\mathrm{term}(T_v)] = C[r\sigma_m] .$$

As $\mathrm{term}(S) = \mathrm{term}(V)$ by Lemma 3.6, $\mathrm{term}(S) = C[l\sigma_m] \to_{\mathcal{R},p} C[r\sigma_m] = \mathrm{term}(T)$ follows.

Finally, consider the direction from left to right. For this suppose $s = C[l\sigma] \to_{\mathcal{R},p} C[r\sigma] = t$ where the position of the hole in C is p. Suppose $S \lhd_p^! U \blacktriangleright_p^! V$ for $\mathrm{term}(S) = s$. We prove that there exists T such that $V \longrightarrow_{\mathcal{G}(\mathcal{R}),p} T$ and $\mathrm{term}(T) = t$. Note that V is $\blacktriangleright_p$-minimal and, as observed above, it is also $\lhd_p$-minimal. Let $v \in V$ be the node corresponding to p, by Lemma 4.13 the node v is unshared. Next, observe $l\sigma = s|_p = \mathrm{term}(S \restriction p) = \mathrm{term}(V \restriction v)$ since $\mathrm{term}(S) = \mathrm{term}(V)$ (c.f. Lemma 3.6). Additionally, Lemma 4.14 reveals $V \restriction v \in \triangledown(l\sigma)$. Further, by Lemma 4.3 we see

$$s = C[l\sigma] = \mathrm{term}(V) = \mathrm{term}(V[V \restriction v]_v) = \mathrm{term}(V[\Box]_v)[l\sigma] .$$

Since the position of the hole in C and $\mathrm{term}(V[\Box]_v)$ coincides, we conclude $C = \mathrm{term}(V[\Box]_v)$.

Let $L \to R \in \mathcal{G}(\mathcal{R})$ be the rule corresponding to $(l \to r) \in \mathcal{R}$, let $(L' \to R') \cong (L \to R)$ be a renaming for V. As $L' \in \vartriangle(l)$ and $V \restriction v \in \triangledown(l\sigma)$, by Lemma 4.5 there exists a morphism $m \colon L' \to V \restriction v$ and hence $V \longrightarrow_{\mathcal{G}(\mathcal{R}),p} T$ for $T = V[m(R')]_v$. Note that for the induced substitution σ_m and $x \in \mathcal{V}\mathrm{ar}(l)$, $\sigma_m(x) = \sigma(x)$. Set $T_v := (m(R') \cup V) \restriction \mathrm{rt}(m(R'))$, hence $T = V[T_v]_v$ and moreover $r\sigma = r\sigma_m = \mathrm{term}(T_v)$ follows as in the first half of the proof. Employing Lemma 4.3 we obtain

$$t = C[r\sigma] = \mathrm{term}(V[\Box]_v)[r\sigma] = \mathrm{term}(V[T_v]_v) = \mathrm{term}(T) .$$

∎

We define $S \twoheadleftarrow\!\!\!\twoheadrightarrow_{\mathcal{G}(\mathcal{R}),p} T$ if and only if $S \lhd_p^! \cdot \blacktriangleright_p^! U \longrightarrow_{\mathcal{G}(\mathcal{R}),p} T$. Employing this notion we can rephrase the conclusion of the Adequacy Theorem as: $s \to_{\mathcal{R},p} t$ if and only if $S \twoheadleftarrow\!\!\!\twoheadrightarrow_{\mathcal{G}(\mathcal{R}),p} T$ for $\mathrm{term}(S) = s$ and $\mathrm{term}(T) = t$.

5. Implementing Term Rewriting Efficiently

Opposed to term rewriting, graph rewriting induces linear size growth in the length of derivations. The latter holds as a single step $\longrightarrow_{\mathcal{G}}$ admits constant size growth:

Lemma 5.1. *If $S \longrightarrow_{\mathcal{G}} T$ then $|T| \leqslant |S| + \Delta$ for some $\Delta \in \mathbb{N}$ depending only on $\mathcal{G}$.*

Proof. Set $\Delta := \max\{|R| \mid (L \to R) \in \mathcal{G}\}$ and the lemma follows by definition. ∎

It is easy to see that a graph rewrite step $S \longrightarrow_{\mathcal{G}} T$ can be performed in time polynomial in the size of the term graph S. By the above lemma we obtain that S can be normalised in time polynomial in $|S|$ and the length of derivations. In the following, we prove a result similar to Lemma 5.1 for the relation $\twoheadleftarrow\!\!\rightarrow_{\mathcal{G}}$, where (restricted) folding and unfolding is incorporated. The main obstacle is that due to unfolding, size growth of $\twoheadleftarrow\!\!\rightarrow_{\mathcal{G}}$ is not bound by a constant in general. We now investigate the relation $\lhd_p$ and $\blacktriangleright_p$.

Lemma 5.2. *Let S be a term graph and let p be a position in S.*

1) *If $S \lhd_p^\ell T$ then $\ell \leqslant |p|$ and $|T| \leqslant |S| + |p|$.*
2) *If $S \blacktriangleright_p^\ell T$ then $\ell \leqslant |S \upharpoonright p|$ and $|T| \leqslant |S|$.*

Proof. We consider the first assertion. For term graphs U, let $P_U = \{w \mid \mathcal{P}os_U(w) = \{q\}$ and $q \leqslant p\}$ be the set of unshared nodes above p. Consider $U \lhd_p V$. Observe that $P_U \subset P_V$ holds: By definition $U \sqsubset_v^u V$ where $\mathcal{P}os_V(u) = \{q\}$ with $q \leqslant p$. Clearly, $P_U \subseteq P_V$, but moreover $u \in P_V$ whereas $u \notin P_U$. Hence for $(S \lhd_p^\ell T) = S = S_0 \lhd_p \dots \lhd_p S_\ell = T$, we observe $P_S = P_{S_0} \subset \dots P_{S_\ell} = P_T$. Note that $|P_S| \geqslant 1$ since $\mathrm{rt}(S) \in P_s$. Moreover, $|P_T| = |p| + 1$ since the node corresponding to p in T is unshared (c.f. Lemma 4.13). Thus from $P_{S_i} \subset P_{S_{i+1}}$ $(0 \leqslant i < \ell)$ we conclude $\ell \leqslant |p|$. Next, we see $|T| \leqslant |S|+|p|$ as $|T| = |S|+\ell$ by definition of $\lhd_p$.

Finally, the second assertion can be proved as above, where we employ that $U \lhd_p V$ implies $|V| = |U| - 1$, c.f. the technical report [4]. ∎

By combining the above two lemmas we derive the following:

Lemma 5.3. *If $S \twoheadleftarrow\!\!\rightarrow_{\mathcal{G}} T$ then $|T| \leqslant |S| + \mathrm{dp}(S) + \Delta$ and $\mathrm{dp}(T) \leqslant \mathrm{dp}(S) + \Delta$ for some $\Delta \in \mathbb{N}$ depending only on $\mathcal{G}$.*

Proof. Consider $S \twoheadleftarrow\!\!\rightarrow_{\mathcal{G}} T$, i.e., $S \lhd_p^! U \blacktriangleright_p^! V \longrightarrow_{\mathcal{G}} T$ for some position p and term graphs U and V. Lemma 5.2 reveals $|U| \leqslant |S| + |p|$ and further $|V| \leqslant |U|$ for $\Delta := \max\{|R| \mid (L \to R) \in \mathcal{G}\}$. As $|p| \leqslant \mathrm{dp}(S)$ we see $|V| \leqslant |S| + \mathrm{dp}(S)$. Since $V \longrightarrow_{\mathcal{G}} T$ implies $|T| \leqslant |V| + \Delta$ (c.f. Lemma 5.1) we establish $|T| \leqslant |S| + \mathrm{dp}(S) + \Delta$. Finally, $\mathrm{dp}(T) \leqslant \mathrm{dp}(S) + \Delta$ follows from the easy observation that both $U \lhd_p V$ and $U \blacktriangleright_p V$ imply $\mathrm{dp}(U) = \mathrm{dp}(V)$, likewise $V \longrightarrow_{\mathcal{G}} T$ implies $\mathrm{dp}(T) \leqslant V + \Delta$. ∎

Lemma 5.4. *If $S \twoheadleftarrow\!\!\rightarrow_{\mathcal{G}}^\ell T$ then $|T| \leqslant (\ell + 1)|S| + \ell^2\Delta$ for $\Delta \in \mathbb{N}$ depending only on $\mathcal{G}$.*

Proof. We prove the lemma by induction on ℓ. The base case follows trivially, so suppose the lemma holds for ℓ, we establish the lemma for $\ell+1$. Consider a derivation $S \twoheadleftarrow\!\!\rightarrow_{\mathcal{G}}^\ell T \twoheadleftarrow\!\!\rightarrow_{\mathcal{G}} U$. By induction hypothesis, $|T| \leqslant (\ell+1)|S| + \ell^2\Delta$. Iterative application of Lemma 5.3 reveals $\mathrm{dp}(T) \leqslant \mathrm{dp}(S) + \ell\Delta$. Thus

$$
\begin{aligned}
|U| &\leqslant |T| + \mathrm{dp}(T) + \Delta \\
&\leqslant \big((\ell+1)|S| + \ell^2\Delta\big) + \big(\mathrm{dp}(S) + \ell\Delta\big) + \Delta \leqslant (\ell+2)|S| + (\ell+1)^2\Delta \ .
\end{aligned}
$$

∎

In the sequel, we prove that an arbitrary graph rewrite step $S \twoheadleftarrow\!\!\rightarrow T$ can be performed in time cubic in the size of S. Lemma 5.4 then allows us to lift the bound on steps to a polynomial bound on derivations in the size of S and the length of derivations. We closely follow the notions of [12]. As model of computation we use k-tape *Turing Machines* (TM for short) with dedicated input- and output-tape. If not explicitly mentioned otherwise, we will use deterministic TMs. We say that a (possibly nondeterministic) TM computes a

relation $R \subseteq \Sigma^* \times \Sigma^*$ if for all $(x,y) \in R$, on input x there exists an accepting run such that y is written on the output tape.

We fix a *standard encoding* for term graphs S. We assume that for each function symbol $f \in \mathcal{F}$ a corresponding tape-symbols is present. Nodes and variables are represented by natural numbers, encoded in binary notation and possibly padded by zeros. We fix the invariant that natural numbers $\{1,\dots,|S|\}$ are used for nodes and variables in the encoding of S. Thus variables (nodes) of S are representable in space $\mathrm{O}(\log(|S|))$. Finally, term graphs S are encoded as a list of *node specifications*, i.e., triples of the form $\langle v, \mathrm{L}(v), \mathrm{Succ}(v)\rangle$ for all $v \in S$ (see [18, Section 13.3]). For a suitable encoding of tuples and lists, a term graph S is representable in size $\mathrm{O}(\log(|S|) * |S|)$. For this, observe that the length of $\mathrm{Succ}(v)$ is bound by the maximal arity of the fixed signature $\mathcal{F}$. In this spirit, we define the *representation size* of a term graph S as $\|S\| := \mathrm{O}(\log(|S|) * |S|)$.

We investigate into the computational complexity of $\lhd_p$ and $\blacktriangleright_p$ first.

Lemma 5.5. *Let S be a term graph and let p a position in S. A term graph T such that $S \lhd_p^! T$ is computable in time* $\mathrm{O}(\|S\|^2)$.

Proof. Suppose $S = S_0 \lhd_p S_1 \lhd_p \cdots \lhd_p S_\ell = T$. By Lemma 5.2, $\ell \leqslant |p| \leqslant |S| \leqslant \|S\|$. One verifies that S_{i+1} is computable from S_i $(0 \leqslant i < \ell)$ in time linear in S_i, and thus linear in S (compare Lemma 5.2). From this it is easy to see that there exists a deterministic TM operating in time quadratic in $\|S\|$ (c.f. the technical report [4]).

∎

Lemma 5.6. *Let S be a term graph and p a position in S. The term graph T such that $S \blacktriangleright_p^! T$ is computable in time* $\mathrm{O}(\|S\|^2)$.

Proof. Define the *height* $ht_U(u)$ of a node u in a term graph U inductively as usual: $ht_U(u) := 0$ if $\mathrm{Succ}(u) = []$ and $ht_U(v) := 1 + \max_{v \in \mathrm{Succ}(u)} ht_U(v)$ otherwise. We drop the reference to the graph U in $ht_U(u)$ in the analysis of the normalising sequence $S \blacktriangleright_p^! T$ below. This is justified as the height of nodes remain stable under $\sqsupset$-reductions.

Recall the definition of $\blacktriangleright_p$: $U \blacktriangleright_p V$ if there exist nodes u, v strictly below p with $U \sqsupset_v^u V$. Clearly, for u, v given, the graph V is constructable from U in time linear in $|U|$. However, finding arbitrary nodes u and v such that $U \sqsupset_v^u V$ takes time quadratic in $|U|$ worst case. Since up to linearly many $\sqsupset$-steps in $|S|$ need to be performed, a straightforward implementation admits cubic runtime complexity. To achieve a quadratic bound in the size of the starting graph S, we construct a TM that implements a bottom up reduction-strategy. More precisely, the machine implements the maximal sequence

$$S = S_1 \sqsupset_{u_1}^! S_2 \sqsupset_{u_2}^! \cdots \sqsupset_{u_{\ell-1}}^! S_\ell \qquad \text{(a)}$$

satisfying, for all $1 \leqslant i < \ell - 1$, (i) either $ht(u_i) = ht(u_{i+1})$ and $u_i \prec u_{i+1}$ or $ht(u_i) < ht(u_{i+1})$, and (ii) for $S_i \sqsupset_{u_i}^{v_{i,1}} \dots \sqsupset_{u_i}^{v_{i,k}} S_{i+1}$, u_i and $v_{i,j}$ $(1 \leqslant j \leqslant k)$ are strictly below p.

By definition $S \blacktriangleright_p^* S_\ell$, it remains to show that the sequence (a) is normalising, i.e., S_ℓ is $\blacktriangleright_p$-minimal. Set $d := \mathrm{dp}(S \upharpoonright p)$ and define, for $0 \leqslant h \leqslant d$,

$$\sqsupset_{(h)} := \bigcup_{u,v \in S \upharpoonright p \wedge ht(v) = h} \sqsupset_v^u .$$

Observe that each $\sqsupset_{u_i}$-step in the sequence (a) corresponds to a step $\sqsupset_{(h)}$ for some $0 \leqslant h \leqslant d$. Moreover, it is not difficult to see that

$$S = S_{i_0} \sqsupset_{(0)}^! S_{i_1} \sqsupset_{(1)}^! \cdots \sqsupset_{(d)}^! S_{i_{d+1}} = S_\ell \qquad \text{(b)}$$

for $\{S_{i_0}, \dots, S_{i_d+1}\} \subseteq \{S_1, \dots, S_{\ell-1}\}$.

Next observe $S_i \sqsupset_{(h_1)} S_{i+1} \sqsupset_{(h_2)} S_{i+2}$ and $h_1 > h_2$ implies $S_i \sqsupset_{(h_2)} \cdot \sqsupset_{(h_1)} S_{i+2}$: suppose $S_i \sqsupset_u^{u'} S_{i+1} \sqsupset_v^{v'} S_{i+2}$ where $ht(u) > ht(v)$ and $u', u, v, v' \in S \upharpoonright p$, we show $S_i \sqsupset_v^{v'} \cdot \sqsupset_u^{u'} S_{i+2}$. Inspecting the proof of Lemma 4.10 we see $\sqsubset_u^{u'} \cdot \sqsupset_v^{v'} \subseteq \sqsupset_v^{v'} \cdot \sqsubset_u^{u'}$ for the particular case that u', u, v and v' pairwise distinct. The latter holds as $ht(u') = ht(u) \neq ht(v) = ht(v')$. Hence it remains to show $S_i \sqsupset_v^{v'} S'_{i+1}$ for some term graph S'_{i+1}, or equivalently $\mathrm{L}_{S_i}(v) = \mathrm{L}_{S_i}(v')$ and $\mathrm{Succ}_{S_i}(v) = \mathrm{Succ}_{S_i}(v')$ by Lemma 4.9. The former equality is trivial, for the latter observe $ht(u') = ht(u) > ht(v) = ht(v')$ and thus neither $u' \notin \mathrm{Succ}_{S_i}(v')$ nor $u' \notin \mathrm{Succ}_{S_i}(v)$. We see $\mathrm{Succ}_{S_i}(v) = \mathrm{Succ}_{S_{i+1}}(v) = \mathrm{Succ}_{S_{i+1}}(v') = \mathrm{Succ}_{S_i}(v')$.

Now suppose that S_ℓ is not $\blacktriangleright_p$-minimal, i.e, $S_\ell \sqsupset_{(h)} U$ for some $0 \leqslant h \leqslant d$ and term graph U. But then we can permute steps in the reduction (b) such that $S_{i_{h+1}} \sqsupset_{(h)} V$ for some term graph V. This contradicts $\sqsupset_{(h)}^!$-minimality of $S_{i_{h+1}}$. We conclude that S_ℓ is $\blacktriangleright_p$-minimal. Thus sequence (a) is $\blacktriangleright_p$-normalising.

Finally, using the derivation (a) it is not difficult to show that there exists a TM operating in time $\mathrm{O}(\|S\|^2)$ that, on input S and p, computes T such that $S \blacktriangleright_p T$. For the construction we kindly refer the reader to the technical report [4]. ∎

Lemma 5.7. *Let S be a term graph, let p be a position of S and let $L \to R$ be a rewrite rule of the simulating graph rewrite system. It is decidable in time $\mathrm{O}(\|S\|^2 * 2^{\|L \to R\|})$ whether $S \longrightarrow_{p, L \to R} T$ for some term graph T. Moreover, the term graph T is computable from S, p and $L \to R$ in time $\mathrm{O}(\|S\|^2 * 2^{\|L \to R\|})$.*

Proof. For the first assertion we can use the *matching*-algorithm as described in [3, Lemma 24]. Based on the morphism returned by this procedure, it is easy to construct a TM that computes the graph T under the stated bound, c.f. the technical report [4]. ∎

Lemma 5.8. *Let S be a term graph and let $\mathcal{G}(\mathcal{R})$ be the simulating graph rewrite system of $\mathcal{R}$. If S is not a normal-form of $\mathcal{G}(\mathcal{R})$ then there exists a position p and rule $(L \to R) \in \mathcal{G}(\mathcal{R})$ such that a term graph T with $S \twoheadleftarrow\!\!\!\rightarrow_{\mathcal{G}(\mathcal{R}), p, L \to R} T$ is computable in time $\mathrm{O}(\|S\|^3)$.*

Proof. The TM searches for a rule $(L \to R) \in \mathcal{G}$ and position p such that $S \twoheadleftarrow\!\!\!\rightarrow_{\mathcal{G}(\mathcal{R}), p, L \to R} T$ for some term graph T. For this, it enumerates the rules $(L \to R) \in \mathcal{G}$ on a separate working tape. For each rule $L \to R$, each node $u \in S$ and some $p \in \mathcal{P}os_S$ it computes S_1 such that $S \blacktriangleright_p^! S_1$ in time quadratic in $\|S\|$ (c.f. Lemma 5.6). Using the machine of Lemma 5.7, it decides in time $2^{\mathrm{O}(\|L\|)} * \mathrm{O}(\|S_1\|^2)$ whether rule $L \to R$ applies to S_1 at position p. Since $\mathcal{R}$ is fixed, $2^{\mathrm{O}(\|L\|)}$ is constant, thus the TM decides whether rule $L \to R$ applies in time $\mathrm{O}(\|S_1\|^2) = \mathrm{O}(\|S\|^2)$. Note that the choice of $p \in \mathcal{P}os_S(u)$ is irrelevant, since $S \blacktriangleright_{p_i}^! S_1$ and $S \blacktriangleright_{p_j}^! S_2$ for $p_i, p_j \in \mathcal{P}os_S(u)$ implies $S_1 \cong S_2$. Hence the node corresponding to p_i in S_1 is a redex with respect to $L \to R$ if and only if the node corresponding to p_j is. Suppose rule $L \to R$ applies at $S_1 \upharpoonright p$. One verifies $S_1 \upharpoonright p \cong S_2 \upharpoonright p$ for term graph S_2 such that $S \lhd_p^! \cdot \blacktriangleright_p^! S_2$. We conclude $S \twoheadleftarrow\!\!\!\rightarrow_{\mathcal{G}(\mathcal{R}), p, L \to R} T$ for some position p and rule $(L \to R) \in \mathcal{G}(\mathcal{R})$ if and only if the above procedure succeeds. From u one can extract some position $p \in \mathcal{P}os_S(u)$ in time quadratic in $\|S\|$. This can be done for instance by implementing the function $\mathsf{pos}(u) = \varepsilon$ if $u = \mathrm{rt}(S)$ and $\mathsf{pos}(u) = pi$ for some node $v \in S$ with $v \xrightharpoonup{i} u$ and $\mathsf{pos}(v) = p$. Overall, the position $p \in \mathcal{P}os_S$ and rule $(L \to R) \in \mathcal{G}$ is found if and only if $S \twoheadleftarrow\!\!\!\rightarrow_{p, L \to R} T$ for some term graph T. Since $|S| \leqslant \|S\|$, and only a constant number of rules have to be checked, the overall runtime is $\mathrm{O}(\|S\|^3)$.

To obtain T from S, p, and $L \to R$, the machine now combines the machines from Lemma 5.5, Lemma 5.6 and Lemma 5.7. These steps can even be performed in time $\mathrm{O}(\|S\|^2)$, employing that the size of intermediate graphs is bound linear in the size of S (compare Lemma 5.2) and that sizes of $(L \to R) \in \mathcal{G}(\mathcal{R})$ are constant. ∎

Lemma 5.9. *Let S be a term graph and let $\ell := \mathrm{dl}(S, \twoheadrightarrow_{\mathcal{G}(\mathcal{R})})$. Suppose $\ell = \Omega(|S|)$.*

1) *Some normal-form of S is computable in deterministic time $\mathrm{O}(\log(\ell)^3 * \ell^7)$.*
2) *Any normal-form of S is computable in nondeterministic time $\mathrm{O}(\log(\ell)^2 * \ell^5)$.*

Proof. We prove the first assertion. Consider the normalising derivation

$$S = T_0 \twoheadrightarrow_{\mathcal{G}(\mathcal{R})} \dots \twoheadrightarrow_{\mathcal{G}(\mathcal{R})} T_l = T \qquad (\dagger)$$

where, for $0 \leqslant i < l$, T_i is obtained from T_{i+1} as given by Lemma 5.8. By Lemma 5.4, we see $|T_i| \leqslant (\ell + 1)|S| + \ell^2\Delta = \mathrm{O}(\ell^2)$. Here the latter equality follows by the assumption $\ell = \Omega(|S|)$. Recall $\|T_i\| = \mathrm{O}(\log(|T_i|) * |T_i|)$ $(0 \leqslant i < l)$ and hence $\|T_i\| = \mathrm{O}(\log(\ell^2) * \ell^2) = \mathrm{O}(\log(\ell) * \ell^2)$. From this, and Lemma 5.8, we obtain that T_{i+1} is computable from T_i in time $\mathrm{O}(\|T_i\|^3) = \mathrm{O}(\log(\ell)^3 * \ell^6)$. Since $l \leqslant \mathrm{dl}(S, \twoheadrightarrow_{\mathcal{G}(\mathcal{R})}) = \ell$ we conclude the first assertion.

We now consider the second assertion. Reconsider the proof of Lemma 5.8. For a given rewrite-position p, a step $S \twoheadrightarrow_{\mathcal{G}(\mathcal{R})} T$ can be performed in time $\mathrm{O}(\|S\|^2)$. A nondeterministic TM can guess some position p, and verify whether the node corresponding to p is a redex in time $\mathrm{O}(\|S\|^2)$. In total, the reduct T can be obtained in nondeterministic time $\mathrm{O}(\|S\|^2)$. Hence, following the proof of the first assertion, one easily verifies the second assertion. ∎

6. Closing the Gap

In this section we state a classification of deterministic as well as nondeterministic polytime-computation based on runtime complexity analysis of rewrite systems. We define semantics of TRSs as follows.

Definition 6.1. Let $\mathcal{N} \subseteq \mathcal{V}\mathrm{al}$ be a finite set of *non-accepting patterns*. We call a term t *accepting* (with respect to $\mathcal{N}$) if there exists no $p \in \mathcal{N}$ such that $p\sigma = t$ for some substitution σ. We say that $\mathcal{R}$ *computes the relation* $R \subseteq \mathcal{V}\mathrm{al} \times \mathcal{V}\mathrm{al}$ with respect to $\mathcal{N}$ if there exists $\mathsf{f} \in \mathcal{D}$ such that for all $s, t \in \mathcal{V}\mathrm{al}$,

$$s \mathrel{R} t :\Longleftrightarrow \mathsf{f}(s) \to^!_{\mathcal{R}} t \text{ and } t \text{ is accepting .}$$

On the other hand, we say that a relation R is computed by $\mathcal{R}$ if R is defined by the above equations with respect to *some* set $\mathcal{N}$ of non-accepting patterns.

For the case that $\mathcal{R}$ is *confluent* we also say that $\mathcal{R}$ computes the (partial) *function* induced by the relation R. Note that the restriction to binary relations is a non-essential simplification. The assertion that for normal-forms t, t is accepting amounts to our notion of *accepting run* of a TRS $\mathcal{R}$. The restriction that $\mathcal{N}$ is finite is essential for the simulation results below: If we implement the computation of $\mathcal{R}$ on a TM, then we also have to be able to effectively test whether t is accepting.

We contrast Definition 6.1 to the way how semantics is given to TRSs in [6]. Basically, in [6] the result of a computation is defined as the *maximal* normal-form with respect to some order on terms. So even non-confluent TRSs compute functions. This definition serves the purpose of characterising *optimisation problems*. In particular, restricted polynomial

interpretations are used to characterise the class of functions OptP as introduced in [13]. Intuitively, an OptP function is computed by a TM with an NP oracle that outputs the *maximal* result of all accepting computation branches. It is not our intention to capture optimisation problems. Instead, we show below a tight correspondence between polynomial runtime-complexity and the class of *functional problems* (FNP for short) associated with NP (see below for a definition). (Note that $\mathsf{FNP} \subseteq \mathsf{OptP}$ and it is expected that the inclusion is strict, see [13, 16].)

First, we show that polynomial runtime-complexity implies polytime computability of the relations defined in the sense of Definition 6.1. For this, we encode terms as graphs and perform rewriting on graphs instead.

Theorem 6.2. *Let $\mathcal{R}$ be a terminating TRS, moreover suppose $\mathrm{rc}_{\mathcal{R}}(n) = \mathrm{O}(n^k)$ for all $n \in \mathbb{N}$ and some $k \in \mathbb{N}$, $k \geqslant 1$. The relations computed by $\mathcal{R}$ are computable in nondeterministic time $\mathrm{O}(n^{5k+2})$. Further, if $\mathcal{R}$ is confluent then the functions computed by $\mathcal{R}$ are computable in deterministic time $\mathrm{O}(n^{7k+3})$.*

Proof. We investigate the complexity of a relation R computed by $\mathcal{R}$. For that, single out the corresponding defined function symbol f and fix some argument $s \in \mathcal{V}\mathrm{al}$. Suppose the underlying set of non-accepting patterns is $\mathcal{N}$. By definition, $s\ R\ t$ if and only if $\mathsf{f}(s) \to^{!}_{\mathcal{R}} t$ and $t \in \mathcal{V}\mathrm{al}$ is accepting with respect to $\mathcal{N}$. Let S be a term graph such that $\mathrm{term}(S) = \mathsf{f}(s)$. Note that $|S| \leqslant |\mathsf{f}(s)|$ and set $\ell := \mathrm{dl}(S, \twoheadrightarrow_{\mathcal{G}(\mathcal{R})})$. By Theorem 4.15, we obtain $S \twoheadrightarrow^{!}_{\mathcal{G}(\mathcal{R})} T$ where $\mathrm{term}(T) = t$, and moreover, $\ell \leqslant \mathrm{rc}_{\mathcal{R}}(|\mathsf{f}(s)|) = \mathrm{O}(n^k)$. By Lemma 5.9 we see that T is computable from S in nondeterministic time $\mathrm{O}(\log(\ell)^2 * \ell^5) = \mathrm{O}(\log(n^k)^2 * n^{5k}) = \mathrm{O}(n^{5k+2})$. Clearly, we can decide in time linear in $\|T\| = \mathrm{O}(\ell^2) = \mathrm{O}(n^{2k})$ (c.f. Lemma 5.4) whether $\mathrm{term}(T) \in \mathcal{V}\mathrm{al}$, further in time quadratic in $\|T\|$ whether $\mathrm{term}(T)$ is accepting. For the latter, we use the matching algorithm of Lemma 5.7 on the fixed set of non-accepting patterns, where we employ $p\sigma = \mathrm{term}(T)$ if and only if there exists a morphism $m\colon P \to T$ for $P \in \triangle(p)$ (c.f. Lemma 4.5 and Lemma 3.5). Hence overall, the accepting condition can be checked in (even deterministic) time $\mathrm{O}(n^{4k})$. If the accepting condition fails, the TM rejects, otherwise it accepts a term graph T representing t. The machine does so in nondeterministic time $\mathrm{O}(n^{5k+2})$ in total. As s was chosen arbitrarily, we conclude the first half of the theorem.

Finally, the second half follows by identical reasoning, where we use the deterministic TM as given by Lemma 5.9 instead of the nondeterministic one. ∎

Let R be a binary relation that is decidable by some *nondeterministic* TM N. That is, the pair (x, y) is accepted by N if and only if $x\ R\ y$ holds. Furthermore, suppose N operates in time polynomial in the size of x. The *function problems* R_F associated with R is: given x, find *some* y such that $x\ R\ y$ holds. The class FNP is the class of all functional problems defined in the above way, compare [16]. FP is the subclass resulting if we only consider function problems in FNP that can be solved in polynomial time by some deterministic TM. As corollary of Theorem 6.2 we obtain:

Corollary 6.3. *Let $\mathcal{R}$ be a terminating TRS with polynomially bounded runtime complexity. Suppose $\mathcal{R}$ computes the relation R. Then $R_F \in \mathsf{FNP}$ for the function problem R_F associated with R. Moreover, if $\mathcal{R}$ is confluent then $R_F \in \mathsf{FP}$.*

Proof. The nondeterministic TM N as given by Theorem 6.2 (the deterministic TM N, respectively) can be used to decide whether $s\ R\ t$ holds. By the assumptions on $\mathcal{R}$, the runtime of N is bounded polynomially in the size of s. To conclude the corollary it suffices

to observe that the term graph S representing s in proof of Theorem 6.2 can be chosen so that $|S| = \mathrm{O}(|s|)$. ∎

In the terminology of [7], we have shown that the number of rewrite steps is an *invariant cost model* for term rewriting. Not only does it reflect the complexity of a TRS in a very natural way, but in fact it truthfully reflects the complexity of rewriting on the standard computational model in complexity theory, the Turing machine. In [7] our result is proved for orthogonal TRSs and innermost or respectively outermost rewriting. Hence our work can be seen as a direct extension of [7], establishing that neither the restriction to orthogonality nor the use of particular reduction-strategies is essential. Based on [7], Dal Lago and Martini establish in [8] that the number of β-steps constitutes an invariant cost model for the *weak* λ-calculus (here reduction below λ-abstractions are disallowed), when reducing under a *call-by-value* or *call-by-need* reduction strategy. Their approach works by embedding β-steps into the rewrite relation as induced by specific orthogonal TRSs, and analysing the complexity of the latter relation. This raises the question whether our result can be used to extend the work by Dal Lago and Martini on λ-calculus. This is subject to further research.

References

[1] M. Avanzini and G. Moser. Complexity Analysis by Rewriting. In *Proc. of 9th FLOPS*, volume 4989 of *LNCS*, pages 130–146. Springer Verlag, 2008.

[2] M. Avanzini and G. Moser. Dependency Pairs and Polynomial Path Orders. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 48–62. Springer Verlag, 2009.

[3] M. Avanzini and G. Moser. Complexity Analysis by Graph Rewriting. In *Proc. of 11th FLOPS*, LNCS. Springer Verlag, 2010. To appear.

[4] M. Avanzini and G. Moser. Technical report: Complexity Analysis by Graph Rewriting Revisited. *CoRR*, cs/CC/1001.5404, 2010. Available at `http://www.arxiv.org/`.

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

[6] G. Bonfante, A. Cichon, J. 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] 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 Verlag, 2009.

[9] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *JAR*, 40(3):195–220, 2008.

[10] A. Koprowski and J. Waldmann. Arctic Termination ... Below Zero. In *Proc. of 19th RTA*, volume 5117 of *LNCS*, pages 202–216. Springer Verlag, 2008.

[11] M. Korp and A. Middeldorp. Match-bounds revisited. *IC*, 207(11):1259–1283, 2009.

[12] D. C. Kozen. *Theory of Computation.* Springer Verlag, first edition, 2006.

[13] M. W. Krentel. The Complexity of Optimization Problems. In *Proc. of 18th STOC*, pages 69–76. ACM, 1986.

[14] G. Moser and A. Schnabl. The Derivational Complexity Induced by the Dependency Pair Method. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 255–260. Springer Verlag, 2009.

[15] G. Moser, A. Schnabl, and J. Waldmann. Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In *Proc. of 28th FSTTCS*, pages 304–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008.

[16] C. H. Papadimitriou. *Computational Complexity.* Addison Wesley Longman, second edition, 1995.

[17] D. Plump. Essentials of Term Graph Rewriting. *ENTCS*, 51:277–289, 2001.

[18] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracks in Theoretical Computer Science.* Cambridge University Press, 2003.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 49-66
http://rewriting.loria.fr/rta/

ABSTRACT MODELS OF TRANSFINITE REDUCTIONS

PATRICK BAHR

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
URL: http://www.diku.dk/~paba
E-mail address: paba@diku.dk

Abstract. We investigate transfinite reductions in abstract reduction systems. To this end, we study two abstract models for transfinite reductions: a metric model generalising the usual metric approach to infinitary term rewriting and a novel partial order model. For both models we distinguish between a weak and a strong variant of convergence as known from infinitary term rewriting. Furthermore, we introduce an axiomatic model of reductions that is general enough to cover all of these models of transfinite reductions as well as the ordinary model of finite reductions. It is shown that, in this unifying axiomatic model, many basic relations between termination and confluence properties known from finite reductions still hold. The introduced models are applied to term rewriting but also to term graph rewriting. We can show that for both term rewriting as well as for term graph rewriting the partial order model forms a conservative extension to the metric model.

1. Introduction

The study of infinitary term rewriting, introduced by Dershowitz et al. [Der91], is concerned with reductions of possibly infinite length. To formalise the concept of transfinite reductions, a variety of different models were investigated in the last 20 years. The most thoroughly studied model is the metric model, both its weak [Der91] and its strong [Ken95] variant. Other models, using for example general topological spaces [Rod98] or partial orders [Cor93, Blo04], were mostly considered to pursue specific purposes. Within these models many fundamental properties do not depend on the particular structure of terms, e.g. the property that strongly converging reductions in the metric model have countable length. Moreover, when studying these different approaches to transfinite reductions, one realises that they often share many basic properties, e.g. in how reductions can be composed and decomposed.

The purpose of this paper is to study transfinite reductions on an abstract level using several different models. This includes a metric model (Section 5) as well as a novel partial order model (Section 6), each of which induces a weak and a strong variant of convergence. Moreover, we introduce an *axiomatic model* of *transfinite abstract reduction systems* (Section 4) which captures the fundamental properties of transfinite reductions. This model

1998 ACM Subject Classification: F.4.2, F.1.1.
Key words and phrases: infinitary rewriting, metric, partial order, abstract reduction system, axiomatic, term rewriting, graph rewriting.

© Patrick Bahr
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.49

subsumes both variants of the metric and the partial order model, respectively, as well as ordinary finite reductions. In fact, we formulate these more concrete models in terms of the axiomatic model, which simplifies their presentation and their analysis substantially. To illustrate this, we reformulate well-known termination and confluence properties in the unifying axiomatic model and show that this then yields the corresponding standard termination and confluence properties for standard finite term rewriting resp. infinitary term rewriting. Additionally, we also prove that basic relations between these properties known from the finite setting also hold in this more general setting.

Lastly, we briefly mention that our models can be applied to term graph rewriting [Bar87] (Section 7) which yields the first formalisation of infinitary term graph rewriting. Moreover, we show that the partial order model is in fact superior to the metric model, at least for interesting cases like terms and term graphs: It can model convergence as in the metric model but additionally allows to distinguish between different levels of divergence.

Related Work. The idea of investigating transfinite reductions on an abstract level was first pursued by Kennaway [Ken92] by studying strongly convergent reductions in an abstract metric framework similar to ours. In this paper we will show that almost all of Kennaway's positive results (except countability of strong convergence) already hold in our more general axiomatic framework, and that countability already holds for strongly continuous reductions.

Kahrs [Kah07] investigated a more concrete model in which he considered weakly convergent reductions in term rewriting systems parametrised by the metric on terms. Although being parametric in the metric space, the results of Kahrs are tied to term rewriting and are for example not applicable to term graph rewriting [Bah09].

The use of partial orders and their notion of *limit inferior* for transfinite reductions is inspired by Blom [Blo04] who studied strongly convergent reductions in lambda calculus using a partial order and compared this to the ordinary metric model of strongly convergent reductions.

2. Preliminaries

We assume familiarity with the basic theory of ordinal numbers, orders and topological spaces [Kel55], as well as term rewriting [Ter03]. In the following, we briefly recall the most important notions.

Transfinite Sequences. We use $\alpha, \beta, \gamma, \lambda, \iota$ to denote ordinal numbers. A *transfinite sequence* (or simply called *sequence*) S of length α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We use $|S|$ to denote the length α of S. If α is a limit ordinal, then S is called *open*. Otherwise, it is called *closed*. If α is a finite ordinal, then S is called *finite*. Otherwise, it is called *infinite*. For a finite sequence $(a_\iota)_{\iota<n}$, we also write $\langle a_0, a_1, \ldots, a_{n-1}\rangle$.

The *concatenation* $(a_\iota)_{\iota<\alpha} \cdot (b_\iota)_{\iota<\beta}$ of two sequences is the sequence $(c_\iota)_{\iota<\alpha+\beta}$ with $c_\iota = a_\iota$ for $\iota < \alpha$ and $c_{\alpha+\iota} = b_\iota$ for $\iota < \beta$. A sequence S is a *prefix* of a sequence T, denoted $S \leq T$ if there is a sequence S' with $S \cdot S' = T$. The prefix of T of length β is denoted $T|_\beta$. The relation $\leq$ forms a complete semilattice.

Metric Spaces. A pair $(M, \mathbf{d})$ is called a *metric space* if $\mathbf{d}\colon M \times M \to \mathbb{R}_0^+$ is a function satisfying $\mathbf{d}(x,y) = 0$ iff $x = y$ (identity), $\mathbf{d}(x,y) = \mathbf{d}(y,x)$ (symmetry), and $\mathbf{d}(x,z) \leq \mathbf{d}(x,y) + \mathbf{d}(y,z)$ (triangle inequality), for all $x,y,z \in M$. If $\mathbf{d}$ instead of the triangle inequality, satisfies the stronger property $\mathbf{d}(x,z) \leq \max\{\mathbf{d}(x,y), \mathbf{d}(y,z)\}$ (strong triangle), $(M, \mathbf{d})$ is called an *ultrametric space.* If a sequence $(a_\iota)_{\iota<\alpha}$ in a metric space converges to an element a, we write $\lim_{\iota\to\alpha} a_\iota$ to denote a. A sequence $(a_\iota)_{\iota<\alpha}$ in a metric space is called *Cauchy* if, for any $\varepsilon \in \mathbb{R}^+$, there is a $\beta < \alpha$ such that, for all $\beta < \iota < \iota' < \alpha$, we have that $\mathbf{d}(m_\iota, m_{\iota'}) < \varepsilon$. A metric space is called *complete* if each of its non-empty Cauchy sequences converges.

Partial Orders. A *partial order* $\leq$ on a class A is a binary relation on A that is *transitive*, *reflexive*, and *antisymmetric*. A partial order $\leq$ on A is called a *complete semilattice* if it has a *least element*, every *directed subset* D of A has a *least upper bound* (*lub*) $\bigsqcup D$ in A, and every subset of A having an upper bound in A also has a least upper bound in A. Hence, complete semilattices also admit a *greatest lower bound* (*glb*) $\bigsqcap B$ for every *non-empty* subset B of A. In particular, this means that for any non-empty sequence $(a_\iota)_{\iota<\alpha}$ in a complete semilattice, its *limit inferior*, defined by $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha}\left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$, always exists. A partial order is called a *linear order* if $a \leq b$ or $b \leq a$ holds for each pair of elements a, b. A linearly ordered subclass of a partially ordered class is also called a *chain.*

Term Rewriting Systems. Instead of finite terms, we consider the set $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ of *infinitary terms* over some *signature* Σ and a countably infinite set $\mathcal{V}$ of variables. We consider $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ as a superset of the set $\mathcal{T}(\Sigma, \mathcal{V})$ of *finite terms.* For a term $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ we use the notation $\mathcal{P}(t)$ to denote the *set of positions* in t. For terms $s, t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ and a position $\pi \in \mathcal{P}(t)$, we write $t|_\pi$ for the *subterm* of t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s.

On $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ a distance function $\mathbf{d}$ can be defined by $\mathbf{d}(s,t) = 0$ if $s = t$ and $\mathbf{d}(s,t) = 2^{-k}$ if $s \neq t$, where k is the minimal depth at witch s and t differ. The pair $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \mathbf{d})$ is known to form a complete ultrametric space [Arn80]. *Partial terms*, i.e. terms over signature $\Sigma_\perp = \Sigma \uplus \{\perp\}$, can be endowed with a relation $\leq_\perp$ by defining $s \leq_\perp t$ iff s can be obtained from t by replacing some subterm occurrences in t by $\perp$. The pair $(\mathcal{T}^\infty(\Sigma_\perp, \mathcal{V}), \leq_\perp)$ is known to form a complete semilattice [Kah93].

A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ such that all variables in r are contained in l. Note that this notion of a TRS is standard in infinitary rewriting [Ken03], but deviates from standard TRSs as it allows infinitary terms on the right-hand side of rules.

As in the finitary case, every TRS $\mathcal{R}$ defines a rewrite relation $\to_\mathcal{R}$:

$$s \to_\mathcal{R} t \iff \exists \pi \in \mathcal{P}(s), l \to r \in R, \sigma\colon\ s|_\pi = l\sigma, t = s[r\sigma]_\pi$$

We write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π.

3. Abstract Reduction Systems

In order to analyse transfinite reductions on an abstract level, we consider *abstract reduction systems (ARS)*. In ARSs, the principal items of interest are the reduction steps of the system. Therefore, the structure of the individual objects on which the reductions are performed is neglected. This abstraction is usually modelled by a pair (A, R) consisting of a set A of objects and a binary relation R on A describing the possible reductions on the objects. The ARS induced by a TRS $\mathcal{R}$ is then simply the pair $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), R)$ with $(s,t) \in R$ iff $s \to_\mathcal{R} t$.

In the setting of infinitary rewriting, however, this model is not appropriate. Instead, we need a model which *reifies* the reduction steps of the system since the semantics of transfinite reductions does not only depend on the objects involved in the reduction but also on *how* each reduction step is performed – at least when we consider *strong convergence*. However, it is not always possible to reconstruct how a reduction was performed given only the starting and end object of it due to so-called *syntactic accidents* [Lév78]: Consider the term rewrite rule $\rho\colon\ f(x) \to x$ and the term $f(f(x))$. The rule ρ can be applied both at root position $\langle\rangle$ and at position $\langle 0 \rangle$ of $f(f(x))$. In both cases the resulting term is $f(x)$.

Therefore, we rather choose a model in which reduction steps are "first-class citizens" [Ter03] similarly to morphisms in a category:

Definition 3.1 (abstract reduction system)**.** An *abstract reduction system (ARS)* $\mathcal{A}$ is a quadruple $(A, \Phi, \mathsf{src}, \mathsf{tgt})$ consisting of a set of *objects* A, a set of *reduction steps* Φ, and *source* and *target functions* $\mathsf{src}\colon\ \Phi \to A$ and $\mathsf{tgt}\colon\ \Phi \to A$, respectively. We write $\varphi\colon\ a \to_\mathcal{A} b$ whenever there are $\varphi \in \Phi$, $a, b \in A$ such that $\mathsf{src}(\varphi) = a$ and $\mathsf{tgt}(\varphi) = b$.

In order to define the semantics of a TRS in terms of an ARS we only need to define an appropriate notion of a reduction step:

Definition 3.2 (operational semantics of TRSs)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS. The ARS *induced* by $\mathcal{R}$, denoted $\mathcal{A}_\mathcal{R}$, is given by $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt})$, where $\Phi = \{(s, \pi, \rho, t) \mid s \to_{\pi,\rho} t\}$, $\mathsf{src}(\varphi) = s$ and $\mathsf{tgt}(\varphi) = t$, for each $\varphi = (s, \pi, \rho, t) \in \Phi$.

A *reduction* in this setting is simply a sequence of reduction steps in an ARS such that consecutive steps are "compatible":

Definition 3.3 (reduction)**.** A sequence $S = (\varphi_\iota)_{\iota<\alpha}$ of reduction steps in an ARS $\mathcal{A}$ is called a *reduction* if there is a sequence of objects $(a_\iota)_{\iota<\widehat{\alpha}}$ in the underlying set A, where $\widehat{\alpha} = \alpha$ if S is open, and $\widehat{\alpha} = \alpha + 1$ if S is closed, such that $\varphi_\iota\colon\ a_\iota \to a_{\iota+1}$ for all $\iota < \alpha$. For such a sequence, we also write $(\varphi\colon\ a_\iota \to a_{\iota+1})_{\iota<\alpha}$ or simply $(a_\iota \to a_{\iota+1})_{\iota<\alpha}$. The reduction S is said to start in a_0, and if S is closed, it is said to end in a_α. If S is finite, we write $S\colon\ a_0 \to^*_\mathcal{A} a_\alpha$. We use the notation $\mathsf{Red}(\mathcal{A})$ to refer to the class of all *non-empty reductions* in $\mathcal{A}$.

Observe that the empty sequence $\langle\rangle$ is always a reduction, and that $\langle\rangle$ starts and ends in a for every object a of the ARS. Also note that this notion of reductions alone does only make sense for sequences of length at most ω. For longer reductions, the ω-th step is not related to the preceding steps of the reduction:

Example 3.4. In the TRS consisting of the rules $a \to f(a)$ and $b \to g(b)$ the following constitutes a valid reduction of length $\omega \cdot 2$:

$$S\colon\ a \to f(a) \to f(f(a)) \to f(f(f(a))) \to\ \dots\ b \to g(b) \to g(g(b)) \to g(g(g(b))) \to\ \dots$$

The second half of the reduction is completely unrelated to the first half. The reason for this issue is that the ω-th reduction step $b \to g(b)$ has no immediate predecessor.

The above problem can occur for all reduction steps indexed by a limit ordinal. For successor ordinals, this is not a problem as by Definition 3.3 the $(\iota+1)$-st step is required to start in the object that the ι-th step ends in. Meaningful definitions for reductions of length beyond ω have to include an appropriate notion of *continuity* which bridges the gaps caused by limit ordinals. Exploring different variants of such a notion of continuity is the topic of the subsequent sections.

4. Transfinite Abstract Reduction Systems

In the last section we have seen that we need a notion of continuity in order to obtain a meaningful model of transfinite reductions. In this section we introduce an axiomatic framework for convergence in which we can derive a corresponding notion of continuity.

The resulting notion of continuity is quite natural and resembles the definition of continuity of real-valued functions: A reduction is *continuous* if every proper prefix *converges* to the object the subsequent suffix is starting in. In order to use this idea, we need to endow an ARS with a notion of convergence:

Definition 4.1 (transfinite abstract reduction system)**.** A *transfinite abstract reduction system (TARS)* $\mathcal{T}$ is a tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathsf{conv})$, such that

(i) $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ is an ARS, called the *underlying ARS* of $\mathcal{T}$, and
(ii) $\mathsf{conv}\colon \mathsf{Red}(\mathcal{A}) \rightharpoonup A$ is a partial function, called *notion of convergence*, which satisfies the following two axioms:

$$\mathsf{conv}(\langle\varphi\rangle) = \mathsf{tgt}(\varphi) \quad \text{for all } \varphi \in \Phi \qquad (\textsc{step})$$

$$\begin{aligned}&\mathsf{conv}(S) = a \text{ and } \mathsf{conv}(T) = b \iff \mathsf{conv}(S \cdot T) = b\\ &\text{for all } a, b \in A,\ S, T \in \mathsf{Red}(\mathcal{A}) \text{ with } T \text{ starting in } a.\end{aligned} \qquad (\textsc{concatenation})$$

That is, we require convergence to include single reduction steps and to be preserved by both composition and decomposition.

Axiom (CONCATENATION) is, in fact, quite comprehensive. But we can split it up into two axioms whose conjunction is equivalent to it:

$$\mathsf{conv}(S) = a \implies \mathsf{conv}(S \cdot T) = \mathsf{conv}(T) \qquad (\textsc{composition})$$

$$\mathsf{conv}(S \cdot T) \text{ defined} \implies \mathsf{conv}(S) = a \qquad (\textsc{continuity})$$

where S and T range over reductions in $\mathsf{Red}(\mathcal{A})$ with T starting in $a \in A$.

Axiom (COMPOSITION) states that the composition of reductions preserves the convergence behaviour whereas (CONTINUITY) ensures that every notion of convergence already includes continuity. To see the latter we need to define convergence and continuity in TARSs:

Definition 4.2 (convergence, continuity)**.** Let $\mathcal{T} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathsf{conv})$ be a TARS and $S \in \mathsf{Red}(\mathcal{T})$ a non-empty reduction starting in $a \in A$. S is said to *converge* to $b \in A$, written $S\colon a \twoheadrightarrow_{\mathcal{T}} b$, if $\mathsf{conv}(S) = b$. S is said to be *continuous*, written $S\colon a \twoheadrightarrow_{\mathcal{T}} \dots$, if for every two $S_1, S_2 \in \mathsf{Red}(\mathcal{T})$ with $S = S_1 \cdot S_2$, we have that S_1 converges to the object S_2 is starting in. If S is continuous but not converging, then S is called *divergent*. For the empty reduction $\langle\rangle$, we define to have $\langle\rangle\colon a \twoheadrightarrow_{\mathcal{T}} a$ and $\langle\rangle\colon a \twoheadrightarrow_{\mathcal{T}} \dots$ for all $a \in A$, i.e. $\langle\rangle$ is always

convergent and continuous. To indicate the length α of a reduction we use the notation $\twoheadrightarrow^{\alpha}_{\mathcal{T}}$. For some object $a \in A$, we write $\mathsf{Cont}(\mathcal{T}, a)$ and $\mathsf{Conv}(\mathcal{T}, a)$ to denote the class of all continuous resp. convergent reductions in $\mathcal{T}$ starting in a.

Axiom (CONTINUITY) is equivalent to the statement that every converging reduction is also continuous. That is, only meaningful – i.e. continuous – reductions can be convergent. This is a natural model which is in particular also adopted in the theory of infinitary term rewriting [Ken03].

Returning to Example 3.4, we can see that for S to be continuous the prefix $S|_\omega$ has to converge to b. However, as one might expect, all notions of convergence for TRSs we will introduce in this paper agree on that $S|_\omega$ converges to f^ω.

Since for closed reductions not only does convergence imply continuity, but also the converse holds true, we have the following proposition:

Proposition 4.3 (convergence of closed reductions)**.** *Let $\mathcal{T}$ be a TARS and S a closed reduction in $\mathcal{T}$. Then S is continuous iff S is converging.*

Proof. The "if" direction follows from (CONTINUITY). The "only if" direction is trivial if S is empty and follows from (STEP) if S has length one. Otherwise, S is of the form $T \cdot \varphi$. Since φ is converging by (STEP) and T is converging by (CONTINUITY), S is converging due to (COMPOSITION). ∎

It is obvious from the definition that a well-defined notion of convergence has to include at least all finite (non-empty) reductions. In fact, the trivial notion of convergence which consists of precisely the *finite reductions* is the least notion of convergence w.r.t. set inclusion of its domain:

Definition 4.4 (finite convergence)**.** Let $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ be an ARS. Then the *finite convergence* of $\mathcal{A}$ is the TARS $\mathcal{A}^f = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathsf{conv})$, where conv is defined by $\mathsf{conv}(S) = b$ iff $S\colon\ a \to^*_{\mathcal{A}} b$. That is, $\mathsf{conv}(S)$ is undefined iff S is infinite.

The TARS given above can be easily checked to be well-defined, i.e. conv satisfies the axioms given in Definition 4.1. We then obtain for every reduction S that $S\colon\ a \to^*_{\mathcal{A}} b$ iff $S\colon\ a \twoheadrightarrow_{\mathcal{A}^f} b$. This shows that TARSs merely provide a generalisation of what is considered to be a well-formed reduction.

Defining conv for the finite convergence was simple. In general, however, it is quite cumbersome to define, as a notion of convergence has to already comprise the corresponding notion of continuity, i.e. satisfy (CONTINUITY). We can avoid this by defining for each partial function $\mathsf{conv}\colon\ \mathsf{Red}(\mathcal{A}) \rightharpoonup A$ its *continuous core* $\overline{\mathsf{conv}}\colon\ \mathsf{Red}(\mathcal{A}) \rightharpoonup A$. For each non-empty reduction $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$ in $\mathcal{A}$ we define

$$\overline{\mathsf{conv}}(S) = \begin{cases} \mathsf{conv}(S) & \text{if } \forall 0 < \beta < \alpha \quad \mathsf{conv}(S|_\beta) = a_\beta \\ \text{undefined} & \text{otherwise} \end{cases}$$

We then have the following lemma:

Lemma 4.5 (continuous core)**.** *Let $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ be an ARS and $\mathsf{conv}\colon\ \mathsf{Red}(\mathcal{A}) \rightharpoonup A$ a partial function satisfying* (STEP) *and* (COMPOSITION)*. Then $\overline{\mathsf{conv}}$ satisfies* (STEP) *and* (CONCATENATION)*, i.e. $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}})$ is a TARS.*

Proof. Straightforward. ∎

Next we have a look at transfinite versions of well-known termination and confluence properties. The basic idea for lifting these properties to the setting of transfinite reductions is to replace finite reductions, i.e. $\to^*$, with transfinite reductions, i.e. $\twoheadrightarrow$.

Applied to the properties *confluence* (CR), *normalisation* (WN), and the *unique normal form property w.r.t. reduction* ($\mathsf{UN}_\to$) we obtain the following transfinite properties:

- CR^∞: If $b \twoheadleftarrow a \twoheadrightarrow c$, then $b \twoheadrightarrow d \twoheadleftarrow c$.
- WN^∞: For each a, there is a normal form b with $a \twoheadrightarrow b$.
- $\mathsf{UN}^\infty_\to$: If $b \twoheadleftarrow a \twoheadrightarrow c$ and b, c are normal forms, then $b = c$.

For properties involving convertibility, i.e. $\leftrightarrow^*$, one has to be more careful. The seemingly straightforward formalisation using transfinite reductions in the *symmetric closure* of the underlying ARS does not work since we do not have a notion of convergence for the symmetric closure. Even if we had one, as in the more concrete models that use a metric space or a partial order, the resulting transfinite convertibility relation would not be symmetric [Bah09].

We therefore follow the approach of Kennaway [Ken92]:

Definition 4.6 (transfinite convertibility)**.** Let $\mathcal{T}$ be a TARS, and a, b objects in $\mathcal{T}$. The objects a and b are called *transfinitely convertible*, written $a \twoheadleftarrow\!\!\twoheadrightarrow_{\mathcal{T}} b$, whenever there is a finite sequence of objects $a_0, \dots, a_n$, $n \geq 0$, in $\mathcal{T}$ such that $a_0 = a$, $a_n = b$, and, for each $0 \leq i < n$, we have $a_i \twoheadrightarrow_{\mathcal{T}} a_{i+1}$ or $a_i \twoheadleftarrow_{\mathcal{T}} a_{i+1}$. The minimal n of such a sequence is called the length of $a \twoheadleftarrow\!\!\twoheadrightarrow_{\mathcal{T}} b$.

This definition of transfinite convertibility is in some sense not "fully transfinite": For two objects to be transfinitely convertible, there has to be a transfinite "reduction" which may only finitely often changes its direction. However, with this definition, transfinite convertibility is an equivalence relation as desired, and we can establish an alternative characterisation of CR^∞ analogously to the original finite version:

Proposition 4.7 (alternative characterisation of CR^∞)**.** *Let $\mathcal{T}$ be a TARS.*

$$\mathcal{T} \text{ is } \mathsf{CR}^\infty \iff \text{Whenever } a \twoheadleftarrow\!\!\twoheadrightarrow b, \text{ then } a \twoheadrightarrow c \twoheadleftarrow b.$$

Proof. The argument is the same as for finite reductions: The "if" direction is trivial, and the "only if" direction can be proved by an induction on the length of $a \twoheadleftarrow\!\!\twoheadrightarrow b$. ∎

With the definition of transfinite convertibility in place, we can define the transfinite versions of the *normal form property* (NF) and the *unique normal form property* (UN):

- NF^∞: For each object a and normal form b with $a \twoheadleftarrow\!\!\twoheadrightarrow b$, we have $a \twoheadrightarrow b$.
- UN^∞: All normal forms a, b with $a \twoheadleftarrow\!\!\twoheadrightarrow b$ are identical.

The above definition of NF^∞ differs from that of Kennaway et al. [Ken95] who, instead of $a \twoheadleftarrow\!\!\twoheadrightarrow b$, use $a \twoheadleftarrow c \twoheadrightarrow b$ as the precondition. One can, however, easily show that both definitions are equivalent.

Having these transfinite properties, we can establish some relations between them analogously to the setting of finite reductions:

Proposition 4.8 (confluence properties)**.** *For every TARS, the following implications hold:*

(i) $\mathsf{CR}^\infty \implies \mathsf{NF}^\infty \implies \mathsf{UN}^\infty \implies \mathsf{UN}^\infty_\to$
(ii) WN^∞ & $\mathsf{UN}^\infty_\to \implies \mathsf{CR}^\infty$

Proof. The arguments are the same as for their finite variants. ∎

Also when formulating a transfinite version of the termination property, we have to be careful. In fact, several different formalisations of transfinite termination can be found in the literature [Ken92, Rod98, Klo05].

We suggest a notion of transfinite termination which we belief is a direct generalisation of finite termination. Recall that an object a in an ARS is terminating iff there is no infinite reduction starting in a. From this we can see that for finite reductions, we can make use of infinite reductions as a meta-concept for defining finite termination. A corresponding meta-concept for transfinite reductions is provided by the class $\mathsf{Conv}(\mathcal{T}, a)$ of converging reductions starting in a ordered by the prefix order $\leq$. The analogue of an infinite reduction, which witnesses finite non-termination, is an unbounded chain in $\mathsf{Conv}(\mathcal{T}, a)$, which witnesses transfinite non-termination:

Definition 4.9 (transfinite termination)**.** Let $\mathcal{T}$ be a TARS. An object a in $\mathcal{T}$ is said to be *transfinitely terminating* (SN^∞) if each chain in $\mathsf{Conv}(\mathcal{T}, a)$ has an upper bound in $\mathsf{Conv}(\mathcal{T}, a)$. The TARS $\mathcal{T}$ itself is called *transfinitely terminating* (SN^∞) if every object in $\mathcal{T}$ is.

The following alternative characterisation of SN^∞ will be useful for comparing our definition to other formalisations of SN^∞ in the literature:

Proposition 4.10 (transfinite termination)**.** *An object a in a TARS $\mathcal{T}$ is SN^∞ iff*
(a) $\mathsf{Cont}(\mathcal{T}, a) \subseteq \mathsf{Conv}(\mathcal{T}, a)$*, and*
(b) every chain in $\mathsf{Conv}(\mathcal{T}, a)$ *is a set.*

Proof. Note that (b) is equivalent to the statement that, for every chain C in $\mathsf{Conv}(\mathcal{T}, a)$, there is an upper bound on the length of the reductions in C.

We show the "only if" direction by proving its contraposition: If (a) is violated, then there is a divergent reduction $S\colon a \twoheadrightarrow \dots$. Hence, the set of all proper prefixes of S forms a chain in $\mathsf{Conv}(\mathcal{T}, a)$ which has no upper bound. Consequently, a is not SN^∞. If (b) is violated, transfinite non-termination of a follows immediately.

For the "if" direction, consider an arbitrary chain C in $\mathsf{Conv}(\mathcal{T}, a)$. Because of (b), C has a lub S. For each proper prefix $S' < S$, there has to be an extension $S'' \geq S$ in C. Since S'' is converging, so is S'. Consequently, S is continuous and, therefore, also convergent, due to (a). Hence, S is an upper bound for C in $\mathsf{Conv}(\mathcal{T}, a)$. ∎

The above characterisation shows that there are two different reasons for transfinite non-termination: Diverging reductions and reductions that can be extended indefinitely. This characterisation of termination closely resembles that of Rodenburg [Rod98] which, however, additionally to (a) and instead of (b) requires an upper bound on the length of reductions. This is too restrictive, since an object, in which for each ordinal α a reduction of length α to a normal form starts, is not transfinitely terminating according to Rodenburg's definition.[1] An example witnessing this difference to our definition can be devised straightforwardly.

In order to verify that our formalisation of SN^∞ is appropriate, we have to make sure that it implies WN^∞:

Proposition 4.11 (SN^∞ is stronger than WN^∞)**.** *For every TARS $\mathcal{T}$, it holds that SN^∞ implies WN^∞ for every object in $\mathcal{T}$.*

[1] In fact, in an earlier draft of this paper we adopted Rodenburg's definition. We thank the anonymous referee who pointed out the mentioned issue.

Proof. We prove the contraposition of the implication using Proposition 4.10. For this purpose, let $\mathcal{T}$ be an TARS and a some object in $\mathcal{T}$ that is not WN^∞. We show that then (a) or (b) of Proposition 4.10 is violated. For this purpose, we assume (a) and show that then (b) does not hold. To this end we define a function f on the class On of ordinal numbers such that, for each $\alpha \in \mathsf{On}$, (1) $f(\alpha)$ is a converging reduction of length α starting in a and (2) $f(\alpha)$ is a proper extension of $f(\iota)$ for all $\iota < \alpha$, i.e. $f(\alpha) > f(\iota)$. Hence, the class $\{f(\alpha) \mid \alpha \in \mathsf{On}\}$ is a chain in $\mathsf{Conv}(\mathcal{T}, a)$ which is not a set since f is a bijection from the proper class On to $\{f(\alpha) \mid \alpha \in \mathsf{On}\}$. The construction of f is justified by the principle of transfinite recursion, and the properties (1) and (2) are established by transfinite induction.

For $\alpha = 0$, both (1) and (2) are trivial. Let α be a successor ordinal $\beta + 1$. By induction hypothesis, we have $f(\beta)\colon a \twoheadrightarrow^\beta b$ for some b. Since a is not WN^∞, b cannot be a normal form. Hence, there is a step $\varphi\colon b \to b'$ in $\mathcal{M}$. Define $f(\alpha) = f(\beta) \cdot \langle \varphi \rangle$. That is, $f(\alpha)\colon a \twoheadrightarrow^\alpha b'$ which shows (1). (2) follows from the induction hypothesis since $f(\beta) < f(\alpha)$.

Let α be a limit ordinal. Since, by the induction hypothesis, (2) holds for all $f(\beta)$, we have that $F = \{f(\beta) \mid \beta < \alpha\}$ is a directed set. Hence, $f(\alpha) = \bigsqcup F$ is well-defined. Consequently, all elements in F are proper prefixes of $f(\alpha)$. This shows (2) and, additionally, it shows that $f(\alpha)$ is a reduction of length α starting in a. Since, by the induction hypothesis $f(\beta)$ is converging for each $\beta < \alpha$, we have that $f(\alpha)$ is continuous. Due to (a), $f(\alpha)$ is also convergent, which shows (1). ■

Note that the transfinite properties we have introduced are equivalent to their finite counterpart if we consider the finite convergence of an ARS. This shows that the transfinite properties that we have given here are in fact generalisations of their original finite versions to the setting of TARS. Moreover, all counterexamples known from the finite setting carry over to the setting of transfinite reductions. This means, for example, that the implications shown in Proposition 4.11 and Proposition 4.8 are in fact strict as they are in the setting of finite reductions.

There are also many interrelations between finite properties which do not hold in the transfinite setting. Notable examples are Newman's Lemma and the implication from subcommutativity to confluence. Counterexamples for these and other interrelations are given by Kennaway [Ken92].

5. Metric Model of Transfinite Reductions

The most common model of infinitary term rewriting is based on the complete ultrametric space of $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. One usually distinguishes between two different variants in this context: A weak variant [Der91], which only takes into account the metric space, and a strong variant [Ken95], which stipulates additional restrictions on the applications of rewrite rules in order to obtain a more well-behaved notion of convergence.

At first we introduce the abstract theory of metric reduction systems. Afterwards, we describe how this can be applied to term rewriting.

Definition 5.1 (metric reduction system)**.** A *metric reduction system (MRS)* $\mathcal{M}$ is a tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$, such that

(i) $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ is an ARS, called the *underlying ARS* of $\mathcal{M}$,
(ii) $\mathbf{d}\colon A \times A \to \mathbb{R}^+_0$ is a function such that $(A, \mathbf{d})$ is a metric space,
(iii) $\mathsf{hgt}\colon \Phi \to \mathbb{R}^+$ is a function, called the *height function*, and

(iv) if $\varphi\colon a \to_{\mathcal{A}} b$, then $\mathbf{d}(a,b) \le \mathsf{hgt}(\varphi)$.

If the metric of an MRS $\mathcal{M}$ is an ultrametric, then $\mathcal{M}$ is called an *ultrametric reduction system (URS)*. Furthermore, an MRS is referred to as *complete* if the underlying metric space is complete. We use the notation $\varphi\colon a \to_h b$ to indicate that $\mathsf{hgt}(\varphi) = h$.

The definition of metric reduction systems follows the idea of *metric abstract reduction systems* investigated by Kennaway [Ken92]. The essential difference between our approach and that of Kennaway is the use of abstract reduction systems with reified reduction steps instead of a family of binary relations. Moreover, unlike Kennaway, we do not restrict ourselves to complete ultrametric spaces. This will allow us to distinguish in which circumstances completeness or an ultrametric is necessary and in which not.

Before continuing the discussion of the abstract model, let us have a look at how TRSs fit into it:

Definition 5.2 (MRS semantics of TRSs)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS. The MRS *induced* by $\mathcal{R}$, denoted $\mathcal{M}_{\mathcal{R}}$, is given by $(\mathcal{T}^\infty(\Sigma,\mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$, where $(\mathcal{T}^\infty(\Sigma,\mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt})$ is the ARS $\mathcal{A}_{\mathcal{R}}$ induced by $\mathcal{R}$, $\mathbf{d}$ is the metric on $\mathcal{T}^\infty(\Sigma,\mathcal{V})$, and hgt is defined as

$$\mathsf{hgt}(\varphi) = 2^{-|\pi|}, \text{ where } \varphi\colon t \to_{\pi,\rho} t'.$$

One can easily check that $\mathcal{M}_{\mathcal{R}}$ indeed forms an MRS for each TRS $\mathcal{R}$. In fact, since the metric on $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ is a *complete ultrametric* [Arn80], $\mathcal{M}_{\mathcal{R}}$ is a *complete URS*.

Next we define for each MRS two notions of convergence:

Definition 5.3 (convergence in MRSs)**.** Let $\mathcal{M} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$ be an MRS. The *weak convergence* of $\mathcal{M}$, denoted $\mathcal{M}^w$, is the TARS given by the tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}^w})$, where $\mathsf{conv}^w(S) = \lim_{\iota\to\widehat{\alpha}} a_\iota$ for a reduction $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$. The *strong convergence* of $\mathcal{M}$, denoted $\mathcal{M}^s$, is the TARS given by the tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}^s})$, where $\mathsf{conv}^s(S) = \lim_{\iota\to\widehat{\alpha}} a_\iota$ for a reduction $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ if S is closed or $\lim_{\iota\to\alpha} h_\iota = 0$; otherwise it is undefined.

The notions of convergence defined above yield precisely the weakly converging [Der91] resp. the strongly converging [Ken95] reductions typically considered in the literature on infinitary term rewriting [Ken03].

From the definition we can immediately derive that strong convergence implies weak convergence. Hence, also strong continuity implies weak continuity.

Note that the height function hgt provides an overapproximation $\mathsf{hgt}(\varphi)$ of the real distance $\mathbf{d}(a,b)$ between the objects a, b involved in a reduction step $\varphi\colon a \to b$. Intuitively, speaking, the difference between weak and strong convergence is that, in the latter variant, the underlying sequence of objects $(a_\iota)_{\iota<\widehat{\alpha}}$ has to converge for the overapproximation provided by hgt as well. In fact, if it is a precise approximation, then weak and strong convergence coincide:

Fact 5.4 (equivalence of weak and strong convergence)**.** *Let* $\mathcal{M} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$ *be an MRS with* $\mathsf{hgt}(\varphi) = \mathbf{d}(a,b)$ *for every reduction step* $\varphi\colon a \to b \in \Phi$*. Then for each reduction* S *in* $\mathcal{M}$ *we have*

(i) $S\colon a \twoheadrightarrow_{\mathcal{M}^w} \dots$ *iff* $S\colon a \twoheadrightarrow_{\mathcal{M}^s} \dots$, *and* *(ii)* $S\colon a \twoheadrightarrow_{\mathcal{M}^w} b$ *iff* $S\colon a \twoheadrightarrow_{\mathcal{M}^s} b$.

Proof. We only need to show that conv^s and conv^w coincide for $\mathcal{M}$. For closed reductions this is trivial. Let $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ be an open reduction. If $\mathsf{conv}^w(S)$ is undefined, then so is $\mathsf{conv}^s(S)$. If $\mathsf{conv}^w(S)$ is defined, then the sequence $(a_\iota)_{\iota<\alpha}$ converges and is

therefore Cauchy. Consequently, the sequence $(\mathbf{d}(a_\iota, a_{\iota+1}))_{\iota<\alpha}$ tends to 0 which implies that also $(h_\iota)_{\iota<\alpha}$ tends to 0 as $h_\iota = \mathbf{d}(a_\iota, a_{\iota+1})$ for each $\iota < \alpha$. Thus, $\mathsf{conv}^s(S) = \mathsf{conv}^w(S)$. ∎

It is instructive to see how hgt provides an overapproximation of the distance function for the example of terms: It assumes that the metric distance between redex and contractum is maximal. That is, the height function only provides a precise approximation if every redex has a root symbol different from the one of its contractum as it is the case for the rule $\rho_1\colon\ c \to g(c)$: The reduction $f(c) \to_{\rho_1} f(g(c)) \to_{\rho_1} f(g(g(c))) \to_{\rho_1} \dots$ converges both weakly and strongly to $f(g^\omega)$. For the rule $\rho_2\colon\ f(x) \to f(g(x))$ this is not the case; both redex and contractum have the same root symbol f. The reduction $f(c) \to_{\rho_2} f(g(c)) \to_{\rho_2} f(g(g(c))) \to_{\rho_2} \dots$ now converges weakly to $f(g^\omega)$ but is not strongly converging.

Note that this also shows the need for reifying reduction steps since in a system containing both ρ_1 and ρ_2 a reduction of the shape $f(c) \to f(g(c)) \to f(g(g(c))) \to \dots$ can be strongly convergent or not, depending on which rules are applied. Similarly, with only a single rule $\rho_3\colon\ g(x) \to g(g(x))$ a reduction of the shape $g(c) \to g(g(c)) \to g(g(g(c))) \to \dots$ can be strongly converging or not, depending on *where* ρ_3 is applied.

The reason for considering strong convergence is that it is considerably more well-behaved [Ken95] than weak convergence [Sim04]. However, weak convergence in the systems characterised in Fact 5.4 inherit the nice properties of strong convergence. For TRSs these systems are precisely those for which the root-symbol of each right-hand side is a function symbol different from the root symbol of the corresponding left-hand side.

When dealing with complete URSs, strong convergence can be characterised by the height only:

Proposition 5.5 (strong convergence in complete URSs)**.** *Let $\mathcal{M}$ be a complete URS. Every open strongly continuous reduction $(a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ in $\mathcal{M}$ is strongly convergent iff $(h_\iota)_{\iota<\alpha}$ tends to* 0*.*

Proof. The "only if" direction is immediate from the definition of strong convergence. For the "if" direction, assume a strongly continuous reduction $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ with $\lim_{\iota\to\alpha} h_\iota = 0$. Then $\lim_{\iota\to\alpha} \mathbf{d}(a_\iota, a_{\iota+1}) = 0$ which in turn implies that $(a_\iota)_{\iota<\alpha}$ is Cauchy as $\mathbf{d}$ is an ultrametric. Since we have a complete metric space, this means that $(a_\iota)_{\iota<\alpha}$ converges. From this and $\lim_{\iota\to\alpha} h_\iota = 0$ we can conclude that S is strongly converging. ∎

Having a complete URS is crucial for the "if" direction of Proposition 5.5. If $\mathcal{M}$ it is not a URS, the underlying sequence $(a_\iota)_{\iota<\alpha}$ might not be Cauchy:

Example 5.6. Consider the MRS $\mathcal{M}$ in the complete metric (but not ultrametric) space $(\mathbb{R}, \mathbf{d})$ with reduction steps of the form $a \to_b (a + b)$, for each $a \in \mathbb{R}, b \in \mathbb{R}^+$. More formally, $\mathcal{M}$ is defined by $\mathcal{M} = (\mathbb{R}, \mathbb{R} \times \mathbb{R}^+, \mathsf{src}, \mathsf{tgt}, \mathbf{d}, \mathsf{hgt})$ with $\mathsf{src}((a, b)) = a$, $\mathsf{tgt}((a, b)) = a + b$, and $\mathsf{hgt}((a, b)) = b$ for all $(a, b) \in \mathbb{R} \times \mathbb{R}^+$. We then have the following reduction in $\mathcal{M}$:

$$0 \to_1 1 \to_{\frac{1}{2}} \left(1 + \frac{1}{2}\right) \to_{\frac{1}{3}} \left(1 + \frac{1}{2} + \frac{1}{3}\right) \to_{\frac{1}{4}} \dots$$

This reduction is trivially strongly continuous but not strongly convergent even though the sequence $(\frac{1}{1+i})_{i<\omega}$ of heights tends to 0. It is not even weakly converging since the series $\sum_{k=1}^{\infty} \frac{1}{k}$ is known to be diverging.

On the other hand, if $\mathcal{M}$ is not complete $(a_\iota)_{\iota<\alpha}$ might not converge:

Example 5.7. Consider the TRS $\mathcal{R}$ with the single rule $a \to f(a)$ and the MRS $\mathcal{M}$ which can be obtained from the induced MRS $\mathcal{M}_\mathcal{R}$ by taking $\mathcal{T}(\Sigma, \mathcal{V})$ as the set of objects instead of $\mathcal{T}^\infty(\Sigma, \mathcal{V})$. Then we have the following reduction in $\mathcal{M}$:

$$a \to_1 f(a) \to_{\frac{1}{2}} f(f(a)) \to_{\frac{1}{4}} f(f(f(a))) \to_{\frac{1}{8}} \dots$$

This reduction is trivially strongly continuous but not strongly convergent, even though the sequence $(2^{-i})_{i<\omega}$ of heights tends to 0. The reduction is not even weakly convergent as the sequence $(f^i(a))_{i<\omega}$ does converge to f^ω in the complete ultrametric space $(\mathcal{T}^\infty(\Sigma, \mathcal{V}), \mathbf{d})$ but does not converge in the incomplete ultrametric space $(\mathcal{T}(\Sigma, \mathcal{V}), \mathbf{d})$

From the above characterisation of strong convergence, we can derive the following more general characterisation:

Proposition 5.8 (strong convergence)**.** *Let S be a reduction in an MRS $\mathcal{M}$.*

(i) If S is strongly convergent, then, for any $h \in \mathbb{R}^+$, there are at most finitely many steps in S whose height is greater than h.

(ii) If S is weakly continuous and, for any $h \in \mathbb{R}^+$, there are at most finitely many steps in S whose height is greater than h, then S is strongly continuous. If, additionally, $\mathcal{M}$ is a complete URS, then S is even strongly convergent.

Proof. (i) The proof of Kennaway [Ken92] also works for MRSs.

(ii) Let $S = (a_\iota \to_{h_\iota} a_{\iota+1})_{\iota<\alpha}$ be a reduction in $\mathcal{M}$. Suppose that S is weakly continuous, and that the set $\{\iota \mid h_\iota > h\}$ is finite for each $h \in \mathbb{R}^+$. We have to show that $\lim_{\iota\to\lambda} h_\iota = 0$ for each limit ordinal $\lambda < \alpha$. To this end, let $\varepsilon > 0$. Then choose some h such that $0 < h < \varepsilon$. Since, by hypothesis, the set $\{\iota \mid h_\iota > h\}$ is finite, there is some ordinal $\beta < \lambda$ such that $h_\iota \le h < \varepsilon$ for all $\beta < \iota < \lambda$. Hence, $\lim_{\iota\to\lambda} h_\iota = 0$.

The second part of (ii) is follows from Proposition 4.3 if S is closed. Otherwise it follows from Proposition 5.5. ∎

The restriction to complete URSs in the second part of (ii) is essential as Example 5.6 and Example 5.7 illustrate.

From this proposition, the following corollary follows as shown by Kennaway [Ken92]:

Corollary 5.9 (countable length of strongly convergent reductions)**.** *In an MRS every strongly convergent reduction has countable length.*

As a result of the above corollary, part (b) of Proposition 4.10 is always satisfied for strong convergence. This makes our definition of SN^∞ equivalent to that of Klop and de Vrijer [Klo05], who considered strong convergence only.

By employing an argument similar to the one used by Klop and de Vrijer [Klo05] for the particular case of infinitary term rewriting, we can generalise Corollary 5.9 to strongly *continuous* reductions, provided we have a complete URS.

Proposition 5.10 (countable length of strongly continuous reductions)**.** *Every strongly continuous reduction in a complete URS has countable length.*

This generalises corresponding results of Kennaway [Ken92] and Klop and de Vrijer [Klo05]. The above proposition is not true for weakly continuous (or convergent) reductions as pointed out by Kennaway [Ken92].

6. Partial Order Model of Transfinite Reductions

The metric model of transfinite reductions has rather restrictive notions of convergence. For example, suppose that we have a TRS consisting of the rules

$$f(x, a) \to f(s(x), b), \quad f(x, b) \to f(s(x), a).$$

Then we can construct the reduction

$$f(0, a) \to f(s(0), b) \to f(s(s(0)), a) \to f(s(s(s(0))), b) \to \ \dots$$

which is neither strongly nor weakly convergent in terms of its MRS semantics. The culprit is the second argument of the f symbol which constantly changes between a and b. However, excluding this "flickering", the reduction seems to converge somehow. The investigation of *partial reduction systems* is aimed at formalising this relaxation of the notion of convergence. With this tool we will be able to identify $f(s^\omega, \bot)$ as the limit of the reduction above.

To this end, a partially ordered set is employed rather than a metric space, and the limit construction is replaced by the limit inferior.

Definition 6.1 (partial reduction system)**.** A *partial reduction system (PRS)* $\mathcal{P}$ is a tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \leq, \mathsf{cxt})$ such that

(i) $\mathcal{A} = (A, \Phi, \mathsf{src}, \mathsf{tgt})$ is an ARS, called the *underlying ARS* of $\mathcal{P}$,
(ii) $(A, \leq)$ is a partially ordered set,
(iii) $\mathsf{cxt}\colon \Phi \to A$ is a function, called the *context function*, and
(iv) if $\varphi\colon a \to_{\mathcal{A}} b$, then $\mathsf{cxt}(\varphi) \leq a, b$.

If the partial order $\leq$ is a complete semilattice, then $\mathcal{P}$ is called *complete*. We use the notation $\varphi\colon a \to_c b$ to indicate that $\mathsf{cxt}(\varphi) = c$.

Also this model can be applied to TRSs. Note, however, that we have to add a fresh constant symbol $\bot$ to the signature in order to use the partial order $\leq_\bot$:

Definition 6.2 (PRS semantics of TRSs)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS. The PRS *induced* by $\mathcal{R}$, denoted $\mathcal{P}_\mathcal{R}$, is given by $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt}, \leq_\bot, \mathsf{cxt})$, with $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \Phi, \mathsf{src}, \mathsf{tgt})$ the ARS $\mathcal{A}_{\mathcal{R}'}$ induced by the TRS $\mathcal{R}' = (\Sigma_\bot, R)$, $\leq_\bot$ the usual partial order on $\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$, and cxt defined by

$$\mathsf{cxt}(\varphi) = t[\bot]_\pi, \text{ where } \varphi\colon t \to_{\pi,\rho} t'.$$

One can easily verify that the context function defined for TRSs satisfies the condition $\mathsf{cxt}(\varphi\colon a \to b) \leq a, b$. Since the partial order on terms forms a *complete semilattice*, this means that the PRS $\mathcal{P}_\mathcal{R}$ induced by a TRS $\mathcal{R}$ is always a *complete PRS*.

Definition 6.3 (convergence of PRSs)**.** Let $\mathcal{P} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \leq, \mathsf{cxt})$ be a PRS. The *weak convergence* of $\mathcal{P}$, denoted $\mathcal{P}^w$, is the TARS given by the tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}}^w)$, where $\mathsf{conv}^w(S) = \liminf_{\iota \to \widehat{\alpha}} a_\iota$ for a reduction $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$. The *strong convergence* of $\mathcal{P}$, denoted $\mathcal{P}^s$, is the TARS given by the tuple $(A, \Phi, \mathsf{src}, \mathsf{tgt}, \overline{\mathsf{conv}}^s)$, where, for a reduction $S = (a_\iota \to_{c_\iota} a_{\iota+1})_{\iota<\alpha}$, $\mathsf{conv}^s(S) = a_\alpha$ if α is a successor ordinal, and $\mathsf{conv}^s(S) = \liminf_{\iota\to\alpha} c_\iota$ if α is a limit ordinal.

Since the limit inferior is always defined for complete semilattices, we immediately obtain that for complete PRSs, continuity and convergence coincide. That is, a reduction is weakly (resp. strongly) continuous iff it is weakly (resp. strongly) convergent. This fact is the main motivation for considering the partial order model as an alternative to the metric model. As a consequence, part (a) of Proposition 4.10 is always satisfied for complete PRSs.

Returning to the initial example of this section we can now observe that the given reduction sequence weakly converges to $f(s^\omega, \bot)$ and strongly converges to $\bot$.

This example also illustrates a major difference compared to the metric model: In MRSs strong convergence is defined by restricting weak convergence. Hence, if a reduction is both weakly and strongly converging, the final result is the same and strong convergence implies weak convergence. For PRSs, however, strong convergence and weak convergence are defined differently. As a result, unlike for MRSs, strong convergence does not imply weak convergence. In order to obtain this behaviour we have to consider *total reductions*:

Definition 6.4 (total reduction). Let $\mathcal{P}$ be a PRS and $S = (a_\iota \to a_{\iota+1})_{\iota<\alpha}$ a reduction in $\mathcal{P}$. We say that S is *total* if each element a_ι is maximal w.r.t. the partial order of $\mathcal{P}$. If we write S as $S\colon a_0 \twoheadrightarrow_{\mathcal{P}^w} a_\alpha$ or $S\colon a_0 \twoheadrightarrow_{\mathcal{P}^s} a_\alpha$, i.e. the convergence of the reduction is explicitly stated, we additionally require a_α to be maximal for S to be total.

Proposition 6.5 (strong convergence implies weak convergence). *For every total reduction S in a PRS $\mathcal{P}$, it holds that*

(i) $S\colon a \twoheadrightarrow_{\mathcal{P}^s} \dots$ implies $S\colon a \twoheadrightarrow_{\mathcal{P}^w} \dots$, and that

(ii) $S\colon a \twoheadrightarrow_{\mathcal{P}^s} b$ implies $S\colon a \twoheadrightarrow_{\mathcal{P}^w} b$.

Proof. Let $S = (a_\iota \to_{c_\iota} a_{\iota+1})_{\iota<\alpha}$. We only need to show that $\mathsf{conv}^s(S) = \mathsf{conv}^w(S)$ whenever $\mathsf{conv}^s(S)$ is a maximal object in $\mathcal{P}$. If S is closed, this is trivial. If S is open we have $\mathsf{conv}^s(S) = \liminf_{\iota\to\alpha} c_\iota \leq \liminf_{\iota\to\alpha} a_\iota = \mathsf{conv}^w(S)$ since, by definition, $c_\iota \leq a_\iota$ for each $\iota < \alpha$. Because $\mathsf{conv}^s(S)$ is maximal, we can conclude that $\mathsf{conv}^s(S) = \mathsf{conv}^w(S)$. ∎

Despite this difference to MRSs, the intuition of the distinction between weak and strong convergence remains the same: Like the height in an MRS, the context $\mathsf{cxt}(\varphi)$ in a PRS overapproximates the difference between the objects a, b involved in a reduction step $\varphi\colon a \to b$. More precisely, it underapproximates the shared structure $a \sqcap b$ of a and b, where $a \sqcap b$ denotes the glb of $\{a, b\}$ w.r.t. the partial order of the PRS. This follows from the condition $\mathsf{cxt}(\varphi) \leq a, b$ which implies $\mathsf{cxt}(\varphi) \leq a \sqcap b$. Likewise, weak and strong convergence coincide if the approximation provided by cxt is precise:

Fact 6.6 (equivalence of weak and strong convergence). *Let $\mathcal{P} = (A, \Phi, \mathsf{src}, \mathsf{tgt}, \leq, \mathsf{cxt})$ be a complete PRS with $\mathsf{cxt}(\varphi) = a \sqcap b$ for every reduction step $\varphi\colon a \to b \in \Phi$. Then for each reduction S in $\mathcal{P}$ we have*

$$(i)\ S\colon a \twoheadrightarrow_{\mathcal{P}^w} \dots \text{ iff } S\colon a \twoheadrightarrow_{\mathcal{P}^s} \dots, \quad \text{and} \quad (ii)\ S\colon a \twoheadrightarrow_{\mathcal{P}^w} b \text{ iff } S\colon a \twoheadrightarrow_{\mathcal{P}^s} b.$$

Proof. Analogously to the proof of Fact 5.4 using the observation that $\liminf_{\iota\to\lambda} a_\iota = \liminf_{\iota\to\lambda}(a_\iota \sqcap a_{\iota+1})$ for all open sequences $(a_\iota)_{\iota<\lambda}$ in a complete semilattice. ∎

Again this fact allows us to transfer results for strong convergence [Bah09] to the setting of weak convergence. And as for Fact 5.4 we can derive from Fact 6.6 that weak and strong convergence coincide for TRSs for which the root symbol of each right-hand side is a function symbol different from the root symbol of the corresponding left-hand side.

7. Metric vs. Partial Order Model

The main motivation for the partial order model is to have a more fine-grained notion of convergence. That is, instead of only being able to distinguish converging and diverging reductions, we have intermediate levels between full convergence and full divergence. Since,

in complete PRSs, continuous reductions are always convergent, the final object of a reduction S indicates the "level of convergence" according to the partial order on objects. If it is $\bot$, the least element of the partial order, then S can be considered fully diverging. If it is a maximal element, e.g. in $\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ a term not containing $\bot$, then S is fully converging.

Using this intuition, the partial order model also gives rise to a notion of *meaninglessness*: We can consider an object a of a complete PRS meaningless if there is an open reduction from a converging to $\bot$. In fact, for strong convergence in orthogonal TRSs, this concept of meaninglessness coincides with so-called *root-active terms* [Bah10].

Under certain quite natural conditions [Bah09], metric convergence can be considered as the fragment of partial order convergence that only considers full convergence. Vice versa, partial order convergence is a conservative extension to metric convergence which also allows partial convergence. This is, in fact, the case for TRSs:

Theorem 7.1 (PRS semantics of TRSs extends MRS semantics)**.** *For each TRS $\mathcal{R}$, the following holds for each $c \in \{w, s\}$:*

(i) $S\colon a \twoheadrightarrow_{\mathcal{P}^c_\mathcal{R}} \dots$ is total iff $S\colon a \twoheadrightarrow_{\mathcal{M}^c_\mathcal{R}} \dots$
(ii) $S\colon a \twoheadrightarrow_{\mathcal{P}^c_\mathcal{R}} b$ is total iff $S\colon a \twoheadrightarrow_{\mathcal{M}^c_\mathcal{R}} b$.

It has been shown [Bah09] that also on so-called *term graphs*, a generalisation of terms, an appropriate complete ultrametric and complete semilattice can be defined. These concepts generalise the metric and the partial order on terms and allow to define infinitary term graph rewriting in our models of transfinite reductions. Following the framework of term graph rewriting systems (TGRSs) of Barendregt et al. [Bar87] one can show that, at least for weak convergence, the same relation between the partial order and the metric model can be observed:

Theorem 7.2 (PRS semantics of TGRSs extends MRS semantics)**.** *For each TGRS $\mathcal{R}$, the following holds:*

(i) $S\colon a \twoheadrightarrow_{\mathcal{P}^w_\mathcal{R}} \dots$ is total iff $S\colon a \twoheadrightarrow_{\mathcal{M}^w_\mathcal{R}} \dots$
(ii) $S\colon a \twoheadrightarrow_{\mathcal{P}^w_\mathcal{R}} b$ is total iff $S\colon a \twoheadrightarrow_{\mathcal{M}^w_\mathcal{R}} b$.

8. Conclusions

The axiomatic model of transfinite reductions provides a simple framework to formulate and analyse the more concrete models presented here and is yet powerful enough to establish many of their fundamental properties. Moreover, the equivalence of transfinite properties for finite convergence and their respective finite counterparts provides additional evidence for the appropriateness of the definition of these transfinite properties.

Fact 5.4 and Fact 6.6 suggest that the metric and the partial order model have a considerable similarity in their discrimination between weak and strong convergence. This raises the question whether there is an appropriate abstraction of these two models that, in contrast to the axiomatic model, is also able to distinguish between weak and strong convergence.

Theorems 7.1 and 7.2 indicate that the partial order model is superior to the metric model as it is able to express convergence as the metric model but additionally allows to explore different levels of divergence in the metric model. Moreover, these results allow to make use of well-known properties of metric infinitary term rewriting in order to study partial order infinitary term rewriting. This was used in [Bah10] to establish several properties of partial order infinitary orthogonal term rewriting such as compression and convergence.

The models that we presented here can be, of course, easily applied to higher-order rewriting systems [Ket05]. However, in the metric approach to infinitary lambda-calculus [Ken97] one usually considers various different metrics and it is not clear what the corresponding partial orders are which then admit a higher-order version of Theorem 7.1.

Acknowledgements

I would like to thank Jakob Grue Simonsen and the alert anonymous referees for carefully reading earlier drafts of this paper and providing valuable feedback. Especially, I want to thank Bernhard Gramlich for his support and his challenging questions during my work on my master's thesis which made this work possible.

References

[Arn80] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundam. Inf.*, 3(4):445–476, 1980.

[Bah09] P. Bahr. *Infinitary Rewriting - Theory and Applications*. Master's thesis, Vienna University of Technology, 2009.
URL `http://www.pa-ba.info/?q=pub/master`

[Bah10] P. Bahr. Partial order infinitary term rewriting and böhm trees. In *RTA 2010*. 2010. cf. this proceedings volume.

[Bar87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, Ri. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In Philip C. Treleaven Jaco de Bakker, A. J. Nijman (ed.), *PARLE 1987*, *LNCS*, vol. 259, pp. 141–158. Springer Berlin / Heidelberg, 1987. doi:10.1007/3-540-17945-3_8.
URL `http://www.springerlink.com/content/pw65n058434q1k65/`

[Blo04] S. Blom. An approximation based approach to infinitary lambda calculi. In Vincent van Oostrom (ed.), *RTA 2004*, *LNCS*, vol. 3091, pp. 221–232. Springer Berlin / Heidelberg, 2004. doi:10.1007/b98160.
URL `http://www.springerlink.com/content/4n3gqw43d1bpnldy/`

[Cor93] A. Corradini. Term rewriting in CT_Σ. In Marie-Claude Gaudel and Jean-Pierre Jouannaud (eds.), *TAPSOFT 1993*, *LNCS*, vol. 668, pp. 468–484. Springer Berlin / Heidelberg, 1993. doi:10.1007/3-540-56610-4_83.
URL `http://www.springerlink.com/content/f73r5p2v370220m4/`

[Der91] N. Dershowitz, S. Kaplan, and D.A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theor. Comput. Sci.*, 83(1):71–96, 1991. doi:DOI:10.1016/0304-3975(91)90040-9.
URL `http://www.sciencedirect.com/science/article/B6V1G-45DHJRB-H/2/767b35171dafdfa511dd0463ea25dbdd`

[Kah93] G. Kahn and G.D. Plotkin. Concrete domains. *Theor. Comput. Sci.*, 121(1-2):187–277, 1993. doi:DOI:10.1016/0304-3975(93)90090-G.
URL `http://www.sciencedirect.com/science/article/B6V1G-45FC431-2K/2/6c30777ef97aea14c529418b4d5c5d4a`

[Kah07] S. Kahrs. Infinitary rewriting: meta-theory and convergence. *Acta Inform.*, 44(2):91–121, 2007. doi:10.1007/s00236-007-0043-2.
URL `http://www.springerlink.com/content/gk52386u20857666/`

[Kel55] J.L. Kelley. *General Topology*, *Graduate Texts in Mathematics*, vol. 27. Springer-Verlag, 1955.

[Ken92] R. Kennaway. On transfinite abstract reduction systems. Tech. rep., CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

[Ken95] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Inf. Comput.*, 119(1):18–38, 1995. doi:DOI:10.1006/inco.1995.1075.
URL `http://www.sciencedirect.com/science/article/B6WGK-45NJJYB-4W/2/7d48d04a2fe97d6e9e1fc5179f31a488`

[Ken97] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theor. Comput. Sci.*, 175(1):93–125, 1997. doi:DOI:10.1016/S0304-3975(96)00171-5.
URL http://www.sciencedirect.com/science/article/B6V1G-3SNTKND-25/2/febf61177fc25aa984b58ee4bb574143

[Ken03] R. Kennaway and F.-J. de Vries. Infinitary rewriting. In Terese [Ter03], chap. 12, pp. 668–711.
URL http://amazon.com/o/ASIN/0521391156/

[Ket05] J. Ketema and J. G. Simonsen. Infinitary combinatory reduction systems. In Jürgen Giesl (ed.), *RTA 2005*, *Lecture Notes in Computer Science*, vol. 3467, pp. 438–452. Springer Berlin / Heidelberg, 2005. doi:10.1007/b135673.
URL http://www.springerlink.com/content/h96d6qlmuhbad903/

[Klo05] J.W. Klop and R.C. de Vrijer. Infinitary normalization. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods (eds.), *We Will Show Them! Essays in Honour of Dov Gabbay*, vol. 2, pp. 169–192. College Publications, 2005.

[Lév78] J.-J. Lévy. *Reductions Correctes et Optimales dans le Lambda-Calcul*. Ph.D. thesis, Université Paris, 1978.

[Rod98] P.H. Rodenburg. Termination and confluence in infinitary term rewriting. *J. Symbolic Logic*, 63(4):1286–1296, 1998.
URL http://www.jstor.org/stable/2586651

[Sim04] J.G. Simonsen. On confluence and residuals in Cauchy convergent transfinite rewriting. *Inf. Process. Lett.*, 91(3):141–146, 2004. doi:DOI:10.1016/j.ipl.2004.03.018.
URL http://www.sciencedirect.com/science/article/B6V0F-4CBVNG4-1/2/d5d0f374f89fd62e07d023512a5b3dfe

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edn., 2003.
URL http://amazon.com/o/ASIN/0521391156/

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 67-84
http://rewriting.loria.fr/rta/

PARTIAL ORDER INFINITARY TERM REWRITING AND BÖHM TREES

PATRICK BAHR

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
URL: http://www.diku.dk/~paba
E-mail address: paba@diku.dk

Abstract. We investigate an alternative model of infinitary term rewriting. Instead of a metric, a partial order on terms is employed to formalise (strong) convergence. We compare this partial order convergence of orthogonal term rewriting systems to the usual metric convergence of the corresponding Böhm extensions. The Böhm extension of a term rewriting system contains additional rules to equate so-called root-active terms. The core result we present is that reachability w.r.t. partial order convergence coincides with reachability w.r.t. metric convergence in the Böhm extension. This result is used to show that, unlike in the metric model, orthogonal systems are infinitarily confluent and infinitarily normalising in the partial order model. Moreover, we obtain, as in the metric model, a compression lemma. A corollary of this lemma is that reachability w.r.t. partial order convergence is a conservative extension of reachability w.r.t. metric convergence.

1. Introduction

The study of infinitary term rewriting as a discipline to investigate infinitely long reductions on terms is mostly based on a metric model [Der91]. Other models, using for example general topological spaces [Rod98] or partial orders [Cor93, Blo04], were mainly considered to pursue quite specific purposes. Since in the metric model, even for orthogonal systems, infinitary rewriting lacks a number of important properties such as *compression* and *infinitary confluence* [Sim04], a stricter variant of convergence, so-called *strong convergence* [Ken95] was considered.

However, even strong convergence does not provide infinitary confluence for all orthogonal term rewriting systems and does not admit *complete developments* for arbitrary sets of redexes. This has been resolved by introducing a notion of meaningless terms [Ken99]. Having this notion, a term rewriting system can be augmented with rules which essentially allow rewriting meaningless terms to a fresh constant $\bot$. When starting with an orthogonal system, the resulting system, called *Böhm extension*, is both infinitarily normalising and infinitarily confluent w.r.t. strong convergence.

1998 ACM Subject Classification: F.4.2.
Key words and phrases: infinitary term rewriting, Böhm trees, partial order, confluence, normalisation.

© Patrick Bahr
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.67

In this paper we present a partial order model of strongly convergent reductions. We show that for orthogonal systems, reachability with this notion of convergence is equivalent to reachability according to metric strong convergence of the corresponding Böhm extensions w.r.t. the least set of meaningless terms, the *root-active* terms. As corollaries we thus obtain infinitary confluence and infinitary normalisation of partial order convergence. Moreover, we can show that this model also enjoys the compression property and admits arbitrary complete developments.

Related Work. This study of strong partial order convergence is inspired by Blom [Blo04] who investigated strong partial order convergence in lambda calculus and compared it to strong metric convergence. Similarly to our findings for orthogonal term rewriting systems, Blom has shown for lambda calculus that reachability in the metric model coincides with reachability in the partial order model modulo equating so-called 0-undefined terms.

Also Corradini [Cor93] studied a partial order model. However, he uses it to develop a theory of parallel reductions which allows simultaneous contraction of a set of mutually independent redexes of left-linear rules. To this end, Corradini defines the semantics of redex contraction in a non-standard way by allowing a partial matching of left-hand sides. Our definition of complete developments also provides, at least for orthogonal systems, a notion of parallel reductions but does so using the standard semantics.

2. Preliminaries

We assume the reader to be familiar with the basic theory of ordinal numbers, orders and topological spaces [Kel55], as well as term rewriting [Ter03]. In the following, we briefly recall the most important notions.

Transfinite Sequences. We use $\alpha, \beta, \gamma, \lambda, \iota$ to denote ordinal numbers. A *transfinite sequence* (or simply called *sequence*) S of length α in a set A, written $(a_\iota)_{\iota<\alpha}$, is a function from α to A with $\iota \mapsto a_\iota$ for all $\iota \in \alpha$. We use $|S|$ to denote the length α of S. If α is a limit ordinal, then S is called *open*. Otherwise, it is called *closed*. If α is a finite ordinal, then S is called *finite*. Otherwise, it is called *infinite*.

The *concatenation* $(a_\iota)_{\iota<\alpha} \cdot (b_\iota)_{\iota<\beta}$ of two sequences is the sequence $(c_\iota)_{\iota<\alpha+\beta}$ with $c_\iota = a_\iota$ for $\iota < \alpha$ and $c_{\alpha+\iota} = b_\iota$ for $\iota < \beta$. A sequence S is a (proper) *prefix* of a sequence T, denoted $S \leq T$ (resp. $S < T$), if there is a (non-empty) sequence S' with $S \cdot S' = T$. The prefix of T of length β is denoted $T|_\beta$. The relation $\leq$ forms a complete semilattice.

Let $S = (a_\iota)_{\iota<\alpha}$ be a sequence. A sequence $T = (b_\iota)_{\iota<\beta}$ is called a *subsequence* of S if there is a monotone function $f\colon \beta \to \alpha$ such that $b_\iota = a_{f(\iota)}$ for all $\iota < \beta$. To indicate this, we write S/f for the subsequence T. If $f(\iota) = f(0) + \iota$ for all $\iota < \beta$, then S/f is called a *segment* of S. That is, T is a segment of S iff there are two sequences T_1, T_2 such that $S = T_1 \cdot T \cdot T_2$. We write $S|_{[\beta,\gamma)}$ for the segment S/f, where $f\colon \alpha' \to \alpha$ is the mapping defined by $f(\iota) = \beta + \iota$ for all $\iota < \alpha'$, with α' the unique ordinal with $\gamma = \beta + \alpha'$. Note that in particular $S|_\alpha = S|_{[0,\alpha)}$ for each sequence S and ordinal $\alpha \leq |S|$.

Partial Orders. A *partial order* $\leq$ on a set A is a binary relation on A that is *transitive*, *reflexive*, and *antisymmetric*. A partial order $\leq$ on A is called a *complete semilattice* if it has a *least element*, every *directed subset* D of A has a *least upper bound* (*lub*) $\bigsqcup D$, and every subset of A having an upper bound also has a least upper bound. Hence, complete semilattices also admit a *greatest lower bound* (*glb*) $\bigsqcap B$ for every *non-empty* subset B of A. In particular, this means that for any non-empty sequence $(a_\iota)_{\iota<\alpha}$ in a complete semilattice, its *limit inferior*, defined by $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha}\left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$, always exists.

With the prefix order $\leq$ on sequences we can generalise concatenation to arbitrary sequences of sequences: Let $(S_\iota)_{\iota<\alpha}$ be a sequence of sequences in a common set. The concatenation of $(S_\iota)_{\iota<\alpha}$, written $\prod_{\iota<\alpha} S_\iota$, is recursively defined as the empty sequence ε if $\alpha = 0$, $\left(\prod_{\iota<\alpha'} S_\iota\right)\cdot S_{\alpha'}$ if $\alpha = \alpha' + 1$, and $\bigsqcup_{\gamma<\alpha}\prod_{\iota<\gamma} S_\iota$ if α is a limit ordinal.

Term Rewriting Systems. Unlike in the traditional framework of term rewriting, we consider the set $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ of *infinitary terms* (or simply *terms*) over some *signature* Σ and a countably infinite set $\mathcal{V}$ of variables. The set $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ is defined as the *greatest* set T such that, for each element $t \in T$, we either have $t \in \mathcal{V}$ or $t = f(t_1,\dots,t_k)$, where $k \geq 0$, $f \in \Sigma^{(k)}$, and $t_1,\dots,t_k \in T$. We consider $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ as a superset of the set $\mathcal{T}(\Sigma,\mathcal{V})$ of *finite terms*. For a term $t \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ we use the notation $\mathcal{P}(t)$ to denote the *set of positions* in t. For terms $s,t \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ and a position $\pi \in \mathcal{P}(t)$, we write $t|_\pi$ for the *subterm* of t at π, $t(\pi)$ for the symbol in t at π, and $t[s]_\pi$ for the term t with the subterm at π replaced by s. Two terms s and t are said to *coincide* in a set of positions $P \subseteq \mathcal{P}(s)\cap\mathcal{P}(t)$ if $s(\pi) = t(\pi)$ for all $\pi \in P$. A position is also called an *occurrence* if the focus lies on the subterm at that position rather than the position itself. Two positions π_1,π_2 are called *disjoint* if neither $\pi_1 \leq \pi_2$ nor $\pi_2 \leq \pi_1$.

On $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ a distance function $\mathbf{d}$ can be defined by $\mathbf{d}(s,t) = 0$ if $s = t$ and $\mathbf{d}(s,t) = 2^{-k}$ if $s \neq t$, where k is the minimal depth at witch s and t differ. The pair $(\mathcal{T}^\infty(\Sigma,\mathcal{V}),\mathbf{d})$ is known to form a complete ultrametric space [Arn80]. *Partial terms*, i.e. terms over signature $\Sigma_\perp = \Sigma \uplus \{\perp\}$, can be endowed with a relation $\leq_\perp$ by defining $s \leq_\perp t$ iff s can be obtained from t by replacing some subterm occurrences in t by $\perp$. The pair $(\mathcal{T}^\infty(\Sigma_\perp,\mathcal{V}),\leq_\perp)$ is known to form a complete semilattice [Kah93]. For a partial term $t \in \mathcal{T}^\infty(\Sigma_\perp,\mathcal{V})$ we use the notation $\mathcal{P}_{\setminus\perp}(t)$ and $\mathcal{P}_\Sigma(t)$ for the set $\{\pi \in \mathcal{P}(t) \mid t(\pi) \neq \perp\}$ of non-$\perp$ positions resp. the set $\{\pi \in \mathcal{P}(t) \mid t(\pi) \in \Sigma\}$ of positions of function symbols. To explicitly distinguish them from partial terms, we call terms in $\mathcal{T}^\infty(\Sigma,\mathcal{V})$ *total*.

A *term rewriting system* (TRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set R of *term rewrite rules* of the form $l \to r$ with $l \in \mathcal{T}^\infty(\Sigma,\mathcal{V}) \setminus \mathcal{V}$ and $r \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ such that all variables in r are contained in l. Note that this notion of a TRS deviates slightly from the standard notion of TRSs in the literature on infinitary rewriting [Ken03] in that it allows infinite terms on the left-hand side of rewrite rules! This generalisation will be necessary to accommodate Böhm extensions. TRSs having only finite left-hand sides are called *left-finite*.

A term t is called *linear* if each variable occurs at most once in t. A TRS $\mathcal{R}$ is called *left-linear* if the left-hand side of every rule in $\mathcal{R}$ is linear. A TRS $\mathcal{R}$ is called *orthogonal* if it is left-linear and has no *critical pairs*.

As in the finitary case, every TRS $\mathcal{R}$ defines a rewrite relation $\to_\mathcal{R}$:

$$s \to_\mathcal{R} t \iff \exists \pi \in \mathcal{P}(s), l \to r \in R, \sigma\colon\; s|_\pi = l\sigma, t = s[r\sigma]_\pi$$

We write $s \to_{\pi,\rho} t$ in order to indicate the applied rule ρ and the position π. The subterm $s|_\pi$ is called a *ρ-redex* or simply *redex*, $r\sigma$ its *contractum*, and $s|_\pi$ is said to be *contracted* to $r\sigma$.

3. Metric Infinitary Term Rewriting

In this section we briefly recall the metric model of infinitary term rewriting [Ken95] and some of its properties. We will use the metric model in two ways: Firstly, it will serve as a yardstick to compare the partial order model to. But most importantly, we will use known results for metric infinitary rewriting and transfer them to the partial order model. In order to accomplish the latter, we will make use of Theorem 5.6 which we shall present at the end of Section 5.

At first we have to make clear what a *reduction* in our setting of infinitary rewriting is:

Definition 3.1 (reduction (step))**.** Let $\mathcal{R}$ be a TRS. A *reduction step* φ in $\mathcal{R}$ is a tuple (s, π, ρ, t) such that $s \to_{\pi,\rho} t$; we also write $\varphi\colon s \to_{\pi,\rho} t$. A *reduction* S in $\mathcal{R}$ is a sequence $(\varphi_\iota)_{\iota<\alpha}$ of reduction steps such that there is a sequence $(t_\iota)_{\iota<\widehat{\alpha}}$ of terms, with $\widehat{\alpha} = \alpha$ if S is open, $\widehat{\alpha} = \alpha + 1$ if S is closed, such that $\varphi_\iota\colon t_\iota \to t_{\iota+1}$. If S is finite, we write $S\colon t_0 \to^* t_\alpha$.

Note that this notion of reductions does only make sense for sequences of length at most ω. For longer reductions, the ω-th step is not related to the preceding steps of the reduction. This holds in general for all reductions steps indexed by a limit ordinal. An appropriate definition of a reduction of length beyond ω requires a notion of continuity to bridge the gaps that arise at limit ordinals. In this section we look at the notion of *strong continuity* modelled by the metric on terms. Since we are not interested in weak continuity [Der91] here, we refer to this notion simply as *continuity*, or *m-continuity* to distinguish it from continuity in the partial order model that we will present in Section 5.

It is important to understand that a reduction is a *sequence of reduction steps* rather than just a sequence of terms. This is crucial for a proper definition of strong continuity, which also depends on where contractions take place:

Definition 3.2 (m-continuity/-convergence)**.** Let $\mathcal{R}$ be a TRS and $S = (\varphi_\iota\colon t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ a non-empty reduction in $\mathcal{R}$. The reduction S is called *m-continuous* if $\lim_{\iota\to\lambda} t_\iota = t_\lambda$, and the sequence $(|\pi_\iota|)_{\iota<\lambda}$ of contraction depths tends to infinity for each limit ordinal $\lambda < \alpha$. Provided it is m-continuous, S is said to *m-converge* to t, written $S\colon t_0 \overset{m}{\twoheadrightarrow} t$, if S is closed and $t = t_\alpha$ or if $(|\pi_\iota|)_{\iota<\alpha}$ tends to infinity and $t = \lim_{\iota\to\alpha} t_\iota$. In this case we also say that t is *m-reachable* from t_0. In order to indicate the length of S and the TRS $\mathcal{R}$, we write $S\colon t_0 \overset{m}{\twoheadrightarrow}{}^{\alpha}_{\mathcal{R}} t$. The empty reduction ε is considered m-continuous and m-convergent for any start and end term, i.e. $\varepsilon\colon t \overset{m}{\twoheadrightarrow} \dots$ and $\varepsilon\colon t \overset{m}{\twoheadrightarrow} t$ for all $t \in \mathcal{T}(\Sigma, \mathcal{V})$.

For a reduction to be m-continuous, each open *proper* prefix of the underlying sequence of terms must converge to the term following next in the sequence. Additionally, the depth at where contractions take place has to tend to infinity for each of the reduction's open proper prefixes. In contrast, m-convergence requires the above conditions to hold for *all* open prefixes, i.e. including the whole reduction itself provided it is open. For example, considering the rule $a \to f(a)$, the reduction $g(a) \to g(f(a)) \to g(f(f(a))) \to \dots$ m-converges to the infinite term $g(f^\omega)$. Note that m-convergence implies m-continuity. Hence, only meaningful, i.e. m-continuous, reductions can be m-convergent. On the other hand not every m-continuous reduction is also m-convergent. Having the rule $g(x) \to g(f(x))$

instead, the reduction $g(a) \to g(f(a)) \to g(f(f(a))) \to \dots$ is trivially m-continuous but is now not m-convergent.

If we only want to express that there is some reduction S with, say, $S\colon s \overset{m}{\twoheadrightarrow} t$, then we simply write $s \overset{m}{\twoheadrightarrow} t$. An example for this notation can be seen in the following phrasing of the Compression Lemma [Ken95]:

Theorem 3.3 (Compression Lemma)**.** *For each left-linear, left-finite TRS, $s \overset{m}{\twoheadrightarrow} t$ implies $s \overset{m}{\twoheadrightarrow}^{\leq\omega} t$.*

As an easy corollary we obtain that the final term of an m-converging reduction can be approximated arbitrarily accurately by a finite reduction:

Corollary 3.4 (finite approximation)**.** *Let $\mathcal{R}$ be a left-linear, left-finite TRS and $s \overset{m}{\twoheadrightarrow} t$. Then, for each depth $d \in \mathbb{N}$, there is a finite reduction $s \to^* t'$ such that t and t' coincide up to depth d, i.e. $\mathbf{d}(t, t') < 2^{-d}$.*

Proof. Assume $s \overset{m}{\twoheadrightarrow} t$. By Theorem 3.3, there is a reduction $S\colon s \overset{m}{\twoheadrightarrow}^{\leq\omega} t$. If S is of finite length, then we are done. If $S\colon s \overset{m}{\twoheadrightarrow}^{\omega} t$, then, by m-convergence, there is some $n < \omega$ such that all reductions steps in S after n take place at a depth greater than d. Consider $S|_n\colon s \to^* t'$. It is clear that t and t' coincide up to depth d. ∎

An important difference of m-converging reductions and finite reductions is the confluence of orthogonal systems. In contrast to finite reachability, m-reachability of orthogonal TRSs does not necessarily have the diamond property, i.e. orthogonal systems are confluent but not infinitarily confluent [Ken95]:

Example 3.5 (failure of infinitary confluence)**.** Consider the orthogonal TRS consisting of the *collapsing* rules $\rho_1\colon f(x) \to x$ and $\rho_2\colon g(x) \to x$ and the infinite term $t = g(f(g(f(\dots))))$. We then obtain the reductions $S\colon t \overset{m}{\twoheadrightarrow} g^\omega$ and $T\colon t \overset{m}{\twoheadrightarrow} f^\omega$ by successively contracting all ρ_1- resp. ρ_2-redexes. However, there is no term s such that $g^\omega \overset{m}{\twoheadrightarrow} s \overset{m}{\twoheadleftarrow} f^\omega$ as both g^ω and f^ω can only be rewritten to themselves, respectively.

In the following sections we discuss two different methods for obtaining an appropriate notion of transfinite reachability which actually has the diamond property.

4. Meaningless Terms and Böhm Trees

Meaningless terms, as formalised by Kennaway et al. [Ken99], are terms which can be considered meaningless because, from a term rewriting perspective, they cannot be distinguished from one another and they do not contribute any information to any computation. For orthogonal TRSs, one such set of terms, in fact the least such set, is the set of *root-active* terms [Ken99]:

Definition 4.1 (root-activeness)**.** Let $\mathcal{R}$ be a TRS and $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$. Then t is called *root-active* if for each reduction $t \to^* t'$, there is a reduction $t' \to^* s$ to a redex s. The set of all root-active terms of $\mathcal{R}$ is denoted $\mathcal{RA}_\mathcal{R}$ or simply $\mathcal{RA}$ if $\mathcal{R}$ is clear from the context.

Intuitively speaking, as the name already suggests, root-active terms are terms that can be contracted at the root arbitrarily often, e.g. the terms f^ω and g^ω from Example 3.5.

In this paper we are only interested in this particular set of meaningless terms. So for the sake of brevity we restrict our discussion in this section to $\mathcal{RA}$ instead of the original more general axiomatic treatment by Kennaway et al. [Ken99].

Since, operationally, root-active terms cannot be distinguished from each other it is appropriate to equate them [Ken99]. This can be done by introducing a new constant symbol $\bot$ and making each root-active term equal to $\bot$. By adding rules which enable rewriting root-active terms to $\bot$, this can be encoded into an existing TRS [Ken99]:

Definition 4.2 (Böhm extension). Let $\mathcal{R}$ be a TRS over Σ, and $\mathcal{U} \subseteq \mathcal{T}^\infty(\Sigma, \mathcal{V})$.

(i) A term $t \in \mathcal{T}^\infty(\Sigma, \mathcal{V})$ is called a *$\bot,\mathcal{U}$-instance* of a term $s \in \mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ if t can be obtained from s by replacing each occurrence of $\bot$ in s with some term in $\mathcal{U}$.

(ii) $\mathcal{U}_\bot$ is the set of terms in $\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V})$ that have a $\bot,\mathcal{U}$-instance in $\mathcal{U}$.

(iii) The *Böhm extension* of $\mathcal{R}$ w.r.t. $\mathcal{U}$ is the TRS $\mathcal{B}_{\mathcal{R},\mathcal{U}} = (\Sigma_\bot, R \cup B)$, where

$$B = \{t \to \bot \mid t \in \mathcal{U}_\bot \setminus \{\bot\}\}$$

We write $s \to_{\mathcal{U},\bot} t$ for a reduction step using a rule in B. If $\mathcal{R}$ and $\mathcal{U}$ are clear from the context, we simply write $\mathcal{B}$ and $\to_\bot$ instead of $\mathcal{B}_{\mathcal{R},\mathcal{U}}$ and $\to_{\mathcal{U},\bot}$, respectively.

A reduction that is m-converging in the Böhm extension $\mathcal{B}$ is called *Böhm-converging.* A term t is called *Böhm-reachable* from s if there is a Böhm-converging reduction from s to t.

Note that, for orthogonal TRSs, $\mathcal{RA}$ is closed under substitutions and, hence, so is $\mathcal{RA}_\bot$ [Ken99]. Therefore, whenever $C[t] \to_{\mathcal{RA},\bot} C[\bot]$, we can assume that $t \in \mathcal{RA}_\bot$.

It it at this point where we, in fact, need the generality of allowing infinite terms on the left-hand side of rewrite rules: The additional rules of a Böhm extension allow possibly infinite terms $t \in \mathcal{U}_\bot \setminus \{\bot\}$ on the left-hand side.

Theorem 4.3 (infinitary confluence of Böhm-converging reductions, [Ken99]). *Let $\mathcal{R}$ be an orthogonal, left-finite TRS. Then the Böhm extension $\mathcal{B}$ of $\mathcal{R}$ w.r.t. $\mathcal{RA}$ is infinitarily confluent, i.e. $s_1 \twoheadleftarrow^{m}_{\mathcal{B}} t \twoheadrightarrow^{m}_{\mathcal{B}} s_2$ implies $s_1 \twoheadrightarrow^{m}_{\mathcal{B}} t' \twoheadleftarrow^{m}_{\mathcal{B}} s_2$.*

The lack of confluence for m-converging reductions is resolved in Böhm extensions by allowing (sub-)terms, which where previously not joinable, to be contracted to $\bot$. Returning to Example 3.5, g^ω and f^ω can be rewritten to $\bot$ as both terms are root-active.

Theorem 4.4 (infinitary normalisation of Böhm-converging reductions, [Ken99]). *Let $\mathcal{R}$ be an orthogonal, left-finite TRS. Then the Böhm extension $\mathcal{B}$ of $\mathcal{R}$ w.r.t. $\mathcal{RA}$ is infinitarily normalising, i.e. for each term t there is a $\mathcal{B}$-normal form Böhm-reachable from t.*

This means that each term t of an orthogonal, left-finite TRS $\mathcal{R}$ has a unique normal form in $\mathcal{B}_{\mathcal{R},\mathcal{RA}}$. This normal form is called the *Böhm tree* of t (w.r.t. $\mathcal{RA}$) [Ken99].

5. Partial Order Infinitary Rewriting

In this section we define an alternative model of infinitary term rewriting which uses the partial order on terms to formalise (strong) convergence of transfinite reductions. To this end we will turn to partial terms which, like in the setting of Böhm extensions, have an additional special symbol $\bot$. The result will be a more fine-grained notion of convergence in which, intuitively speaking, a reduction can be diverging in some positions but at the same time converging in other positions. The "diverging parts" are then indicated by a $\bot$-occurrence in the final term of the reduction:

Example 5.1. Consider the TRS consisting of the rules $h(x) \to h(g(x)), c \to g(c)$ and the term $t = f(a, c)$. In this system, we have the reduction

$$S\colon\ f(h(a), b) \to f(h(g(a)), b) \to f(h(g(a)), g(b)) \to f(h(g(g(a))), g(b)) \to \dots$$

which alternately contracts the redex in the left and in the right argument of f.

Reduction S does not m-converge as the depth at which contractions are performed does not tend to infinity. However, this does only happen in the left argument of f, not in the other one. With the notion of p-convergence, we will discover S to be p-converging to the term $f(\bot, g^\omega)$:

Definition 5.2 (p-continuity/-convergence)**.** Let $\mathcal{R} = (\Sigma, R)$ be a TRS and $S = (\varphi_\iota \colon t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ a non-empty reduction in $\mathcal{R}' = (\Sigma_\bot, R)$. The reduction S is called *p-continuous* if $\liminf_{\iota\to\lambda} c_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$, where $c_\iota = t_\iota[\bot]_{\pi_\iota}$. Each c_ι is called the *context* of the reduction step φ_ι. Provided it is p-continuous, S is said to *p-converge* to t, written $S\colon t_0 \xrightarrow{p}\!\!\!\!\!\twoheadrightarrow t$ if S is closed and $t = t_{\alpha+1}$ or if $t = \liminf_{\iota\to\alpha} c_\iota$. In this case we also say that t is *p-reachable* from t_0. In order to indicate the length of S, we write $S\colon t_0 \xrightarrow{p}\!\!\!\!\!\twoheadrightarrow^{\alpha} t$. The empty reduction ε is considered p-continuous and p-convergent for any start and end term.

What makes this notion of p-convergence *strong*, similar to the notion of m-convergence we are considering here, is the choice of taking the contexts $t_\iota[\bot]_{\pi_\iota}$ for defining the limit behaviour of reductions instead of the whole terms t_ι. The context $t_\iota[\bot]_{\pi_\iota}$ provides a conservative underapproximation of the shared structure $t_\iota \sqcap t_{\iota+1}$ of two consecutive terms t_ι and $t_{\iota+1}$. In fact, $t_\iota[\bot]_{\pi_\iota} \le_\bot t_\iota \sqcap t_{\iota+1}$. Returning to Example 5.1, we can observe that with the weaker notion of p-convergence, i.e. using t_ι instead of $t_\iota[\bot]_{\pi_\iota}$ for the limit behaviour, reduction S would p-converge to $f(h(g^\omega), g^\omega)$ instead of $f(\bot, g^\omega)$.

This approach is analogous to the metric notion of strong convergence which requires $|\pi_\iota|$ to tend to infinity, i.e. $2^{-|\pi_\iota|}$ to tend to 0. However, $2^{-|\pi_\iota|}$ is an overapproximation of the actual difference $\mathbf{d}(t_\iota, t_{\iota+1})$ of two consecutive terms t_ι and $t_{\iota+1}$, i.e. $2^{-|\pi_\iota|} \ge \mathbf{d}(t_\iota, t_{\iota+1})$.

Note that we have to consider reductions over the extended signature $\Sigma_\bot$, i.e. reductions containing partial terms. Thus, from now on, we assume reductions in a TRS over Σ to be implicitly over $\Sigma_\bot$. When we want to make it explicit that a reduction S contains only total terms, we say that S is *total*. When we say that $S\colon s \xrightarrow{p}\!\!\!\!\!\twoheadrightarrow t$ is total, we mean that both the reduction S and the final term t are total.[1]

Due to the partial order $\le_\bot$ on partial terms being a complete semilattice, the limit inferior is defined for any sequence of partial terms. Hence, any p-continuous reduction is also p-convergent. This is one of the major differences to m-convergence/-continuity. Nevertheless, p-convergence is a meaningful notion of convergence. The final term of a p-convergent reduction contains a $\bot$ subterm at each position at which the reduction is "locally diverging" as we have seen in Example 5.1. We will call these positions *volatile*:

Definition 5.3 (volatility)**.** Let $\mathcal{R}$ be a TRS and $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\lambda}$ an open p-converging reduction in $\mathcal{R}$. A position π is said to be *volatile* in S if, for each ordinal $\beta < \lambda$, there is some $\beta \le \gamma < \lambda$ such that $\pi_\gamma = \pi$. If π is volatile in S and no proper prefix of π is volatile in S, then π is called *outermost-volatile*.

In Example 5.1 the position 0 is outermost-volatile in the reduction S. One can show that $\bot$ subterms are indeed created precisely at outermost-volatile positions [Bah09]:

Lemma 5.4 ($\bot$ subterms in open reductions)**.** *Let $\mathcal{R}$ be a TRS and $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\lambda}$ an open reduction in $\mathcal{R}$ p-converging to t_λ. Then, for every position π, we have the following:*

(i) If π is volatile in S, then $\pi \notin \mathcal{P}_{\setminus\bot}(t_\lambda)$.

[1]Note that if S is open, the final term t is not explicitly contained in S. Hence, the totality of S does not necessarily imply the totality of t.

(ii) $t_\lambda(\pi) = \bot$ *iff*

(a) π *is outermost-volatile in* S, *or*

(b) there is some $\beta < \lambda$ *such that* $t_\beta(\pi) = \bot$ *and* $\pi_\iota \not\leq \pi$ *for all* $\beta \leq \iota < \lambda$.

(iii) Let t_ι *be total for all* $\iota < \lambda$. *Then* $t_\lambda(\pi) = \bot$ *iff* π *is outermost-volatile in* S.

From this we can deduce that the absence of volatile positions is equivalent to the totality of a p-converging reduction:

Lemma 5.5 (total reductions)**.** *Let* $\mathcal{R}$ *be a TRS,* s *a total term in* $\mathcal{R}$, *and* $S\colon s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$. $S\colon s \overset{p}{\twoheadrightarrow} t$ *is total iff no prefix of* S *has a volatile position.*

Proof. The "only if" direction follows straightforwardly from Lemma 5.4.

We prove the "if" direction by induction on the length of S. If $|S| = 0$, then the totality of S follows from the assumption of s being total. If $|S|$ is a successor ordinal, then the totality of S follows from the induction hypothesis since single reduction steps preserve totality. If $|S|$ is a limit ordinal, then the totality of S follows from the induction hypothesis using Lemma 5.4. ∎

The following theorem is the central tool for transferring results for m-convergent reductions to the realm of p-convergence:

Theorem 5.6 (total p-convergence = m-convergence)**.** *For every reduction* S *in a TRS,* $S\colon s \overset{p}{\twoheadrightarrow} t$ *is total iff* $S\colon s \overset{m}{\twoheadrightarrow} t$.

We won't go into the details of the proof of Theorem 5.6 here but instead refer to [Bah09]. The key for the proof are the following two observations: At first, the limit inferior and the limit of a sequence of total terms coincide whenever the limit exists or the limit inferior is a total term. Secondly, for each open m-converging reduction $S = (\varphi\colon t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\lambda}$ the limit inferior of the sequence of terms $(t_\iota)_{\iota<\lambda}$ coincides with the limit inferior of the sequence of contexts $(t_\iota[\bot]_{\pi_\iota})_{\iota<\lambda}$ since the $\bot$'s in $(t_\iota[\bot]_{\pi_\iota})_{\iota<\lambda}$ are "pushed down" deeper and deeper due to the m-convergence of S.

6. Complete Developments

There are several methods to show (finitary) confluence of orthogonal systems. A quite instructive technique uses notions of *residuals* and *complete developments* [Ter03]. Intuitively speaking, the residuals of a set of redexes are the remains of this set of redexes after a reduction, and a complete development of a set of redexes is a reduction which only contracts residuals of these redexes and ends in a term with no residuals. Kennaway et al. [Ken95] have lifted these notions to (metric) infinitary term rewriting. However, in contrast to the finitary setting, complete developments do not always exists in infinitary orthogonal term rewriting, e.g. for the term f^ω from Example 3.5 and the set of all redex occurrences in it.

In this section we define residuals and complete developments in the setting of partial order infinitary term rewriting and show that complete developments do always exist for orthogonal TRSs and converge to a unique term. Having this, we can show the Infinitary Strip Lemma which is a crucial tool for proving our main result. However, since the proofs of these results are rather technical and tedious we will not provide the full proofs here but rather refer the interested reader to the author's thesis [Bah09] where detailed proofs for all results in this section can be found.

At first we need to formalise the notion of residuals. It is virtually equivalent to the definition for m-convergence by Kennaway et al. [Ken95]:

Definition 6.1 (descendants, residuals)**.** Let $\mathcal{R}$ be a TRS, $S\colon\ t_0 \xrightarrow{p}\!\!\!\!\twoheadrightarrow^{\alpha}_{\mathcal{R}} t_\alpha$, and $U \subseteq \mathcal{P}_{\setminus\bot}(t_0)$. The *descendants* of U by S, denoted $U/\!/S$, is the set of positions in t_α inductively defined as follows:

(a) If $\alpha = 0$, then $U/\!/S = U$.
(b) If $\alpha = 1$, i.e. $S\colon\ t_0 \to_{\pi,\rho} t_1$ for some $\rho\colon\ l \to r$, take any $u \in U$ and define the set R_u as follows: If $\pi \not\leq u$, then $R_u = \{u\}$. If u is in the pattern of the ρ-redex, i.e. $u = \pi \cdot \pi'$ with $\pi' \in \mathcal{P}_\Sigma(l)$, then $R_u = \emptyset$. Otherwise, i.e. if $u = \pi \cdot w \cdot x$, with $l|_w \in \mathcal{V}$, then $R_u = \{\pi \cdot w' \cdot x \,|\, r|_{w'} = l|_w\}$. Define $U/\!/S = \bigcup_{u\in U} R_u$.
(c) If $\alpha = \alpha' + 1$, then $U/\!/S = (U/\!/S|_{\alpha'})/\!/\varphi_{\alpha'}$, where $S = (\varphi_\iota)_{\iota<\alpha}$.
(d) If α is a limit ordinal, then $\quad U/\!/S = \mathcal{P}_{\setminus\bot}(t_\alpha) \cap \liminf_{\iota\to\alpha} U/\!/S|_\iota$
That is, $u \in U/\!/S \quad$ iff $\quad u \in \mathcal{P}_{\setminus\bot}(t_\alpha)$ and $\exists\beta < \alpha\forall\beta \leq \iota < \alpha\colon\ u \in U/\!/S|_\iota$

If, in particular, U is a set of redex occurrences, then $U/\!/S$ is also called the set of *residuals* of U by S. Moreover, by abuse of notation, we write $u/\!/S$ instead of $\{u\}/\!/S$.

Clauses (a), (b) and (c) are as in the finitary setting. Clause (d) lifts the definition to the infinitary setting. However, the only difference to the definition of Kennaway et al. is, that we consider partial terms here. Yet, for technical reasons, the notion of descendants has to be restricted to non-$\bot$ occurrences. Since $\bot$ cannot be a redex, this is not a restriction for residuals, though.

As for finitary rewriting and metric infinitary rewriting, we have that residuals are always redexes and are pairwise disjoint if the original redexes are:

Proposition 6.2 ((disjoint) residuals)**.** *Let $\mathcal{R}$ be an orthogonal TRS, $S\colon\ s \xrightarrow{p}\!\!\!\!\twoheadrightarrow_{\mathcal{R}} t$ and U a set of redex occurrences in s. Then the following holds:*

(i) $U/\!/S$ is a set of redex occurrences in t.
(ii) If the occurrences in U are pairwise disjoint, then so are the occurrences in $U/\!/S$.

The property of residuals being redexes is, in fact, crucial for the concept of complete developments as it requires all residuals to be eventually contracted:

Definition 6.3 ((complete) development)**.** Let $\mathcal{R}$ be an orthogonal TRS, s a partial term in $\mathcal{R}$, and U a set of redex occurrences in s.

(i) A *development* of U in s is a p-converging reduction $S\colon\ s \xrightarrow{p}\!\!\!\!\twoheadrightarrow^{\alpha} t$ in which each reduction step $\varphi_\iota\colon\ t_\iota \to_{\pi_\iota} t_{\iota+1}$ contracts a redex at $\pi_\iota \in U/\!/S|_\iota$ for $\iota < \alpha$.
(ii) A development $S\colon\ s \xrightarrow{p}\!\!\!\!\twoheadrightarrow t$ of U in s is called *complete*, denoted $S\colon\ s \xrightarrow{p}\!\!\!\!\twoheadrightarrow_U t$, if $U/\!/S = \emptyset$.

This is a straightforward generalisation of complete developments known from the finitary setting and coincides with the corresponding formalisation for metric infinitary rewriting [Ken95] if restricted to total terms. However, unlike in the metric setting, partial order infinitary rewriting admits complete developments for any orthogonal system:

Proposition 6.4 (complete developments)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t. Then U has a complete development in t.*

This result follows from the fact that every p-continuous reduction is also p-converging. Proving the above proposition simply amounts to devising a reduction strategy which eventually contracts all redexes. A *parallel-outermost* strategy achieves this.

Next we need to show that the final term of a complete development is uniquely defined by the initial set of redex occurrences U. Using a technique of *paths and jumps* similar to the one described by Kennaway and de Vries [Ken03], we can define for each partial term t,

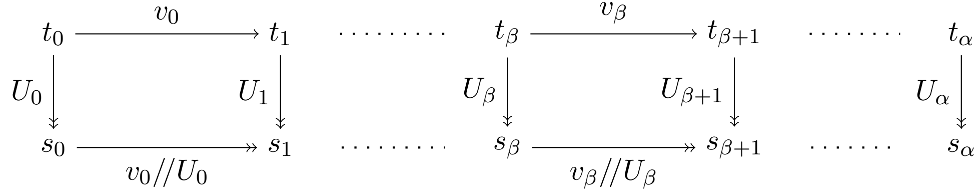


Figure 1: The Infinitary Strip Lemma.

set of redex occurrences U in t, and orthogonal TRS $\mathcal{R}$, a term $\mathcal{F}(t, U, \mathcal{R})$ that is the final term of a complete development of U in t:

Proposition 6.5 (unique p-convergence of complete developments)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t. Then each complete development of U in t p-converges to $\mathcal{F}(t, U, \mathcal{R})$.*

We can use the above result in order to show that descendants by complete developments are uniquely defined. To achieve this, one can use the well-known labelling technique that keeps track of descendants by means of syntactic methods (e.g. see [Ter03]):

Proposition 6.6 (unique descendants of complete developments)**.** *Let $\mathcal{R}$ be an orthogonal TRS, t a partial term in $\mathcal{R}$, and U a set of redex occurrences in t. Then, for each set $V \subseteq \mathcal{P}_{\setminus\bot}(t)$ and two complete developments S and T of U in t, respectively, it holds that $V//S = V//T$.*

As a corollary we obtain that complete developments enjoy the diamond property:

Corollary 6.7 (diamond property of complete developments)**.** *Let $\mathcal{R}$ be an orthogonal TRS and $t \overset{p}{\twoheadrightarrow}_U t_1$ and $t \overset{p}{\twoheadrightarrow}_V t_2$ be two complete developments of U respectively V in t. Then t_1 and t_2 are joinable by complete developments $t_1 \overset{p}{\twoheadrightarrow}_{V//U} t'$ and $t_2 \overset{p}{\twoheadrightarrow}_{U//V} t'$.*

The result of this effort of analysing complete developments is the Infinitary Strip Lemma for p-convergence:

Proposition 6.8 (Infinitary Strip Lemma)**.** *Let $\mathcal{R}$ be an orthogonal TRS, $S\colon\ t_0 \overset{p}{\twoheadrightarrow}^{\alpha} t_\alpha$, and $T\colon\ t_0 \overset{p}{\twoheadrightarrow}_U s_0$ a complete development of a set U of disjoint redex occurrences in t_0. Then t_α and s_0 are joinable by $S/T\colon\ s_0 \overset{p}{\twoheadrightarrow} s_\alpha$ and a complete development $T/S\colon\ t_\alpha \overset{p}{\twoheadrightarrow}_{U//S} s_\alpha$.*

The idea of the construction of S/T and T/S is illustrated in Figure 1. Each U_ι is the set of residuals of U by the reduction $S|_\iota$. Each arrow in the diagram represents a complete development of the indicated set of redex occurrences. In particular, each v_ι indicates the redex occurrence contracted in the ι-th step of S. The construction uses an induction on the length α of the horizontal reduction S. The case $\alpha = 0$ is trivial. For α a successor ordinal, the statement follows from the induction hypothesis using Corollary 6.7. For α a limit ordinal we can make use of the fact that, by Proposition 6.2 each U_ι is a set of pairwise disjoint redex occurrences. The constructed reduction S/T is called the *projection* of S by T. Likewise, T/S is called the *projection* of T by S.

7. p-convergence and Böhm-convergence

In this section we shall show the core result of this paper: For orthogonal, left-finite TRSs, p-reachability and Böhm-reachability w.r.t. $\mathcal{RA}$ coincide. As corollaries of that, leveraging the properties of Böhm-convergence, we obtain both infinitary normalisation and infinitary confluence of orthogonal systems in the partial order model. Moreover, we will show that p-convergence also satisfies the compression property.

The central step of the proof of the equivalence of both models of infinitary rewriting is an alternative characterisation of root-active terms which is captured by the following definition:

Definition 7.1 (destructiveness, fragility)**.** Let $\mathcal{R}$ be a TRS.

(i) A reduction $S\colon t \overset{p}{\twoheadrightarrow} s$ is called *destructive* if ε is a volatile position in S.
(ii) A partial term t in $\mathcal{R}$ is called *fragile* if a destructive reduction starts in t.

Looking at the definition, fragility seems to be a more general concept than root-activeness: A term is fragile iff it admits a reduction in which infinitely often a redex at the root is contracted. For orthogonal TRSs, root-active terms are characterised in almost the same way. The difference is that only total terms are considered and that the stipulated reduction contracting infinitely many root redexes has to be of length ω. However, we shall show the set of total fragile terms to be equal to the set of root-active terms by establishing a compression lemma for destructive reductions.

Using Lemma 5.4 we can immediately derive the following alternative characterisations:

Fact 7.2 (destructiveness, fragility)**.** *Let $\mathcal{R}$ be a TRS.*

(i) A reduction $S\colon s \overset{p}{\twoheadrightarrow} t$ is destructive iff S is open and $t = \bot$
(ii) A partial term t in $\mathcal{R}$ is fragile iff there is an open p-convergent reduction $t \overset{p}{\twoheadrightarrow} \bot$.

Using this, we can establish that any p-convergent reduction can be simulated by a Böhm-convergent reduction w.r.t. total, fragile terms:

Proposition 7.3 (p-reachability implies Böhm-reachability)**.** *Let $\mathcal{R}$ be a TRS, $\mathcal{U}$ the set of fragile terms in $\mathcal{T}^\infty(\Sigma, \mathcal{V})$, and $\mathcal{B}$ the Böhm extension of $\mathcal{R}$ w.r.t. $\mathcal{U}$. Then, for each p-convergent reduction $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$, there is a Böhm-convergent reduction $s \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t$.*

Proof. Assume that there is a reduction $S = (t_\iota \to_{\pi_\iota} t_{\iota+1})_{\iota<\alpha}$ in $\mathcal{R}$ that p-converges to t_α. We will construct an m-convergent reduction $T\colon\ t_0 \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t_\alpha$ in $\mathcal{B}$ by removing reduction steps in S that take place at or below outermost-volatile positions of some prefix of S and replace them by $\to_\bot$-steps.

Let π be an outermost-volatile position of some prefix $S|_\lambda$. Then there is some ordinal $\beta < \lambda$ such that no reduction step between β and λ in S takes place strictly above π, i.e. $\pi_\iota \not< \pi$ for all $\beta \le \iota < \lambda$. Such an ordinal β must exist since otherwise π would not be an outermost-volatile position in $S|_\lambda$. Hence, we can construct a destructive reduction $S'\colon\ t_\beta|_\pi \overset{p}{\twoheadrightarrow} \bot$ by taking the subsequence of the segment $S|_{[\beta,\lambda)}$ that contains the reduction steps at π or below. Note that $t_\beta|_\pi$ might still contain the symbol $\bot$. Since $\bot$ is not relevant for the applicability of rules in $\mathcal{R}$, each of the $\bot$ symbols in $t_\beta|_\pi$ can be safely replaced by arbitrary total terms, in particular by terms in $\mathcal{U}$. Let r be a term that is obtained in this way. Then there is a destructive reduction $S''\colon\ r \overset{p}{\twoheadrightarrow} \bot$ that applies the same rules at the same positions as in S'. Hence, $r \in \mathcal{U}$. By construction, r is a $\bot,\mathcal{U}$-instance of $t_\beta|_\pi$ which means that $t_\beta|_\pi \in \mathcal{U}_\bot$. Additionally, $t_\beta|_\pi \neq \bot$ since there is a non-empty reduction

$S'\colon t_\beta|_\pi \xrightarrow{p}\!\!\!\!\rightarrow \bot$ starting in $t_\beta|_\pi$. Consequently, there is a rule $t_\beta|_\pi \to \bot$ in $\mathcal{B}$. Let T' be the reduction that is obtained from $S|_\lambda$ by replacing the β-th step, which we can assume w.l.o.g. to take place at π, by a step with the rule $t_\beta|_\pi \to \bot$ at the same position π and removing all reduction steps φ_ι taking place at π or below for all $\beta < \iota < \lambda$. Let t' be the term that the reduction T' p-converges to. t_λ and t' can only differ at position π or below. However, by construction, we have $t'(\pi) = \bot$ and, by Lemma 5.4, $t_\lambda(\pi) = \bot$. Consequently, $t' = t_\lambda$.

This construction can be performed for all prefixes of S and their respective outermost-volatile positions. Thereby, we obtain a p-converging reduction $T\colon t_0 \xrightarrow{p}\!\!\!\!\rightarrow_{\mathcal{B}} t_\alpha$ for which no prefix has a volatile position. By Lemma 5.5, T is a total reduction. Note that $\mathcal{B}$ is a TRS over the extended signature $\Sigma' = \Sigma \uplus \{\bot\}$, i.e. terms containing $\bot$ are considered total. Hence, by Theorem 5.6, $T\colon t_0 \xrightarrow{m}\!\!\!\!\rightarrow_{\mathcal{B}} t_\alpha$. ∎

This already provides one direction of the equivalence we want to establish. Before we make the next step, we need the following lemma shown by Kennaway et al. [Ken99]:

Lemma 7.4 (postponement of $\to_\bot$-steps). *Let $\mathcal{R}$ be a left-linear, left-finite TRS and $\mathcal{B}$ some Böhm extension of $\mathcal{R}$. Then $s \xrightarrow{m}\!\!\!\!\rightarrow_{\mathcal{B}} t$ implies $s \xrightarrow{m}\!\!\!\!\rightarrow_{\mathcal{R}} s' \xrightarrow{m}\!\!\!\!\rightarrow_{\bot} t$ for some term s'.*[2]

In the next proposition we show that, excluding $\bot$ subterms, the final term of a p-converging reduction can be approximated arbitrarily well by a finite reduction. This corresponds to Corollary 3.4 which establishes finite approximations for m-convergent reductions.

Proposition 7.5 (finite approximation). *Let $\mathcal{R}$ be a left-linear, left-finite TRS and $s \xrightarrow{p}\!\!\!\!\rightarrow t$. Then, for each finite set $P \subseteq \mathcal{P}_{\setminus\bot}(t)$, there is a reduction $s \to^* t'$ such that t and t' coincide in P.*

Proof. Assume that $s \xrightarrow{p}\!\!\!\!\rightarrow_{\mathcal{R}} t$. Then, by Proposition 7.3, there is a reduction $s \xrightarrow{m}\!\!\!\!\rightarrow_{\mathcal{B}} t$, where $\mathcal{B}$ is the Böhm extension of $\mathcal{R}$ w.r.t. the set of total, fragile terms of $\mathcal{R}$. By Lemma 7.4, there is a reduction $s \xrightarrow{m}\!\!\!\!\rightarrow_{\mathcal{R}} s' \xrightarrow{m}\!\!\!\!\rightarrow_{\bot} t$. Clearly, s' and t coincide in $\mathcal{P}_{\setminus\bot}(t)$. Let $d = \max\{|\pi| \mid \pi \in P\}$. Since P is finite, d is well-defined. By Corollary 3.4, there is a reduction $s \to^*_{\mathcal{R}} t'$ such that t' and s' coincide up to depth d and, thus, in particular they coincide in P. Consequently, since s' and t coincide in $\mathcal{P}_{\setminus\bot}(t) \supseteq P$, t and t' coincide in P, too. ∎

In order to establish a compression lemma for destructive reductions we need that fragile terms are preserved by finite reductions. We can obtain this from the following more general lemma showing that destructive reductions are preserved by forming projections as constructed in the Infinitary Strip Lemma:

Lemma 7.6 (preservation of destructive reductions by projections). *Let $\mathcal{R}$ be an orthogonal TRS, $S\colon t_0 \xrightarrow{p}\!\!\!\!\rightarrow t_\alpha$ a destructive reduction, and $T\colon t_0 \xrightarrow{p}\!\!\!\!\rightarrow_U s_0$ a complete development of a set U of disjoint redex occurrences. Then the projection $S/T\colon s_0 \xrightarrow{p}\!\!\!\!\rightarrow s_\alpha$ is also destructive.*

Proof. We consider the situation depicted in Figure 1. Since $S\colon t_0 \xrightarrow{p}\!\!\!\!\rightarrow t_\alpha$ is destructive, we have, for each $\beta < \alpha$, some $\beta \leq \gamma < \alpha$ such that $v_\gamma = \varepsilon$. If $v_\gamma = \varepsilon$, then also $\varepsilon \in v_\gamma // U_\gamma$ unless $\varepsilon \in U_\gamma$. As by Proposition 6.2, U_γ is a set of pairwise disjoint positions, $\varepsilon \in U_\gamma$ implies $U_\gamma = \{\varepsilon\}$. This means that if $v_\gamma = \varepsilon$ and $\varepsilon \in U_\gamma$, then $U_\iota = \emptyset$ for all $\gamma < \iota < \alpha$. Thus, there is only at most one $\gamma < \alpha$ with $\varepsilon \in U_\gamma$. Therefore, we have, for each $\beta < \alpha$, some $\beta \leq \gamma < \alpha$ such that $\varepsilon \in v_\gamma // U_\gamma$. Hence, T is destructive. ∎

[2] Strictly speaking, if s is not a total term, i.e. it contains $\bot$, then we have to consider the system that is obtained from $\mathcal{R}$ by extending its signature to $\Sigma_\bot$.

As a consequence of this preservation of destructiveness by forming projections, we obtain that the set of fragile terms is closed under finite reductions:

Lemma 7.7 (closure of fragile terms under finite reductions)**.** *In each orthogonal TRS, the set of fragile terms is closed under finite reductions.*

Proof. Let t be a fragile term and $T\colon t \to^* t'$ a finite reduction. Hence, there is a destructive reduction starting in t. A straightforward induction proof on the length of T, using Lemma 7.6, shows that there is a destructive reduction starting in t'. Thus, t' is fragile. ∎

Now we can show that destructiveness does not need more that ω steps in orthogonal, left-finite TRSs. This property will be useful for proving the equivalence of root-activeness and fragility of total terms as well the Compression Lemma for p-convergent reductions.

Proposition 7.8 (Compression Lemma for destructive reductions)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and t a partial term in $\mathcal{R}$. If there is a destructive reduction starting in t, then there is a destructive reduction of length ω starting in t.*

Proof. Let $S\colon t_0 \overset{p}{\twoheadrightarrow}{}^{\lambda} \bot$ be a destructive reduction starting in t_0. Hence, there is some $\alpha < \lambda$ such that $S|_\alpha\colon t_0 \overset{p}{\twoheadrightarrow} s_1$, where s_1 is a ρ-redex for some $\rho\colon l \to r \in R$. Let P be the set of pattern positions of the ρ-redex s_1, i.e. $P = \mathcal{P}_\Sigma(l)$. Due to the left-finiteness of $\mathcal{R}$, P is finite. Hence, by Proposition 7.5, there is a finite reduction $t_0 \to^* s'_1$ such that s_1 and s'_1 coincide in P. Hence, because $\mathcal{R}$ is left-linear, also s'_1 is a ρ-redex. Now consider the reduction $T_0\colon t_0 \to^* s'_1 \to_{\rho,\varepsilon} t_1$ ending with a contraction at the root. T_0 is of finite length and, according to Lemma 7.7, t_1 is fragile.

Since t_1 is again fragile, the above argument can be iterated arbitrarily often which yields for each $i < \omega$ a finite reduction $T_i\colon t_i \to^* t_{i+1}$ whose last step is a contraction at the root. Then the concatenation $T = \prod_{i<\omega} T_i$ of these reductions is a destructive reduction of length ω starting in t_0. ∎

The above proposition bridges the gap between fragility and root-activeness. Whereas the former concept is defined in terms of transfinite reductions, the latter is defined in terms of finite reductions. By Proposition 7.8, however, a fragile term is always finitely reducible to a redex. This is the key to the observation that fragility is not only quite similar to root-activeness but is, in fact, essentially the same concept.

Proposition 7.9 (root-activeness = fragility)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and t a total term in $\mathcal{R}$. Then t is root-active iff t is fragile.*

Proof. The "only if" direction is easy: If t is root-active, then there is a reduction S of length ω starting in t with infinitely many steps taking place at the root. Hence, $S\colon t \overset{p}{\twoheadrightarrow}{}^{\omega} \bot$ is a destructive reduction, which makes t a fragile term.

For the converse direction we assume that t is fragile and show that, for each reduction $t \to^* s$, there is a reduction $s \to^* t'$ to a redex t'. By Lemma 7.7, also s is fragile. Hence, there is a destructive reduction $S\colon s \overset{p}{\twoheadrightarrow} \bot$ starting in s. According to Proposition 7.8, we can assume that S has length ω. Therefore, there is some $n < \omega$ such that $S|_n\colon s \to^* t'$ for a redex t'. ∎

Before we prove the missing direction of the equality of p-reachability and Böhm-reachability we need the property that m-convergent reductions consisting only of $\to_\bot$-steps can be compressed to length at most ω as well:

Lemma 7.10 (compression of $\to_\bot$-steps)**.** *Consider the Böhm extension of an orthogonal TRS w.r.t. its root-active terms and* $S\colon s \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\bot t$ *with* $s \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$, $t \in \mathcal{T}^\infty(\Sigma_\bot,\mathcal{V})$*. Then there is an* m*-converging reduction* $T\colon s \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\bot t$ *of length at most* ω *that is a complete development of a set of disjoint occurrences of root-active terms in* s*.*

Proof. The proof is essentially the same as that of Lemma 7.2.4 from Ketema [Ket06]. ∎

The important part of the above lemma is the statement that only terms in $\mathcal{RA}$ are contracted instead of the general case where a $\to_\bot$ -step contracts a term in $\mathcal{RA}_\bot \supset \mathcal{RA}$.

Finally, we have gathered all tools necessary in order to prove the converse direction of the equivalence of p-reachability and Böhm-reachability w.r.t. root-active terms.

Theorem 7.11 (p-reachability = Böhm-reachability w.r.t. $\mathcal{RA}$)**.** *Let* $\mathcal{R}$ *be an orthogonal, left-finite TRS and* $\mathcal{B}$ *the Böhm extension of* $\mathcal{R}$ *w.r.t. its root-active terms. Then* $s \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} t$ *iff* $s \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\mathcal{B} t$*.*

Proof. The "only if" direction follows immediately from Proposition 7.9 and Proposition 7.3.

Now consider the converse direction: Let $s \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\mathcal{B} t$ be an m-convergent reduction in $\mathcal{B}$. W.l.o.g. we assume s to be total. Due to Lemma 7.4, there is a term $s' \in \mathcal{T}^\infty(\Sigma,\mathcal{V})$ such that there are m-convergent reductions $S\colon s \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\mathcal{R} s'$ and $T\colon s' \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\bot t$. By Lemma 7.10, we can assume that in $s' \xrightarrow{m}\!\!\!\!\twoheadrightarrow_\bot t$ only pairwise disjoint occurrences of root-active terms are contracted. By Proposition 7.9, each root-active term $r \in \mathcal{RA}$ is fragile, i.e. we have a destructive reduction $r \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} \bot$ starting in r. Thus, we can construct a p-converging reduction $T'\colon s' \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} t$ by replacing each step $C[r] \to_\bot C[\bot]$ in T with the corresponding reduction $C[r] \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} C[\bot]$. By combining T' with the m-converging reduction S, which, according to Theorem 5.6, is also p-converging, we obtain the p-converging reduction $S \cdot T'\colon s \xrightarrow{p}\!\!\!\!\twoheadrightarrow_\mathcal{R} t$. ∎

With this equivalence, p-convergent reductions inherit a number of important properties that are enjoyed by Böhm-convergent reductions:

Theorem 7.12 (infinitary confluence)**.** *Every orthogonal, left-finite TRS is infinitarily confluent. That is, for each orthogonal, left-finite TRS,* $s_1 \twoheadleftarrow\!\!\!\!\xleftarrow{p} t \xrightarrow{p}\!\!\!\!\twoheadrightarrow s_2$ *implies* $s_1 \xrightarrow{p}\!\!\!\!\twoheadrightarrow t' \twoheadleftarrow\!\!\!\!\xleftarrow{p} s_2$*.*

Proof. Leveraging Theorem 7.11, this theorem follows from Theorem 4.3. ∎

Returning to Example 3.5 again, we can see that the terms g^ω and f^ω can now be joined by repeatedly contracting the redex at the root which yields destructive reductions $g^\omega \xrightarrow{p}\!\!\!\!\twoheadrightarrow \bot$ and $f^\omega \xrightarrow{p}\!\!\!\!\twoheadrightarrow \bot$, respectively.

Theorem 7.13 (infinitary normalisation)**.** *Every orthogonal, left-finite TRS is infinitarily normalising. That is, for each orthogonal, left-finite TRS* $\mathcal{R}$ *and a partial term* t *in* $\mathcal{R}$*, there is an* $\mathcal{R}$*-normal form* p*-reachable from* t*.*

Proof. This follows immediately from Theorem 7.11 and Theorem 4.4. ∎

Combining Theorem 7.12 and Theorem 7.13, we obtain that each term in an orthogonal TRS has a unique normal form w.r.t. p-convergence. Due to Theorem 7.11, this unique normal form is the Böhm tree w.r.t. root-active terms.

Since p-converging reductions in orthogonal TRS can always be transformed such that they consist of a prefix which is an m-convergent reduction and a suffix consisting of nested destructive reductions, we can employ the Compression Lemma for m-convergent reductions (Theorem 3.3) and the Compression Lemma for destructive reductions (Proposition 7.8) to obtain the Compression Lemma for p-convergent reductions:

Theorem 7.14 (Compression Lemma for p-convergent reductions)**.** *For each orthogonal, left-finite TRS, $s \overset{p}{\twoheadrightarrow} t$ implies $s \overset{p}{\twoheadrightarrow}^{\leq\omega} t$.*

Proof. Let $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$. According to Theorem 7.11, we have $s \overset{m}{\twoheadrightarrow}_{\mathcal{B}} t$ for the Böhm extension $\mathcal{B}$ of $\mathcal{R}$ w.r.t. $\mathcal{RA}$ and, therefore, by Lemma 7.4, we have reductions $S\colon\ s \overset{m}{\twoheadrightarrow}_{\mathcal{R}} s'$ and $T\colon\ s' \overset{m}{\twoheadrightarrow}_{\bot} t$. Due to Theorem 3.3, we can assume S to be of length at most ω and, due to Theorem 5.6, to be p-convergent, i.e $S\colon\ s \overset{p}{\twoheadrightarrow}^{\leq\omega}_{\mathcal{R}} s'$. If T is the empty reduction, then we are done. If not, then T is a complete development of pairwise disjoint occurrences of root-active terms according to Lemma 7.10. Hence, each step is of the form $C[r] \to_{\bot} C[\bot]$ for some root-active term r. By Proposition 7.9, for each such term r, there is a destructive reduction $r \overset{p}{\twoheadrightarrow}_{\mathcal{R}} \bot$ which we can assume, in accordance with Proposition 7.8, to be of length ω. Hence, each step $C[r] \to_{\bot} C[\bot]$ can be replaced by the reduction $C[r] \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} C[\bot]$. Concatenating these reductions results in a reduction $T'\colon\ s' \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ of length at most $\omega \cdot \omega$. If $S\colon\ s \overset{p}{\twoheadrightarrow}^{\leq\omega}_{\mathcal{R}} s'$ is of finite length, we can interleave the reduction steps in T' such that we obtain a reduction $T''\colon\ s' \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} t$ of length ω. Then we have $S \cdot T''\colon\ s \overset{p}{\twoheadrightarrow}^{\omega}_{\mathcal{R}} t$. If $S\colon\ s \overset{p}{\twoheadrightarrow}^{\leq\omega}_{\mathcal{R}} s'$ has length ω, we construct a reduction $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ as follows: As illustrated above, T' consists of destructive reductions taking place at some pairwise disjoint positions. These steps can be interleaved into the reduction S resulting into a reduction $s \overset{p}{\twoheadrightarrow}_{\mathcal{R}} t$ of length ω. The argument for that is similar to that employed in the successor case of the induction proof of the Compression Lemma of Kennaway et al. [Ken95]. ∎

We can use the Compression Lemma for p-convergent reductions to obtain a stronger variant of Theorem 5.6 for orthogonal TRSs:

Corollary 7.15 (m-reachability $=$ p-reachability of total terms)**.** *Let $\mathcal{R}$ be an orthogonal, left-finite TRS and $s, t \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$. Then $s \overset{m}{\twoheadrightarrow} t$ iff $s \overset{p}{\twoheadrightarrow} t$.*

Proof. The “only if” direction follows immediately from Theorem 5.6. For the “if” direction assume a reduction $S\colon\ s \overset{p}{\twoheadrightarrow} t$. According to Theorem 7.14, there is a reduction $T\colon\ s \overset{p}{\twoheadrightarrow}^{\leq\omega} t$. Hence, since s is total and totality is preserved by single reduction steps, $T\colon\ s \overset{p}{\twoheadrightarrow}^{\leq\omega} t$ is total. Applying Theorem 5.6, yields that $T\colon\ s \overset{m}{\twoheadrightarrow}^{\leq\omega} t$. ∎

8. Conclusions

Infinitary term rewriting in the partial order model provides a more fine-grained notion of convergence. Formally, every meaningful, i.e. p-continuous, reduction is also p-converging. Practically, p-converging reductions can end in a term containing $\bot$’s indicating positions of “local divergence”. Theorem 5.6 and Corollary 7.15 indicate that the partial model coincides with the metric model but additionally allows a more detailed inspection of non-m-converging reductions. Instead of the coarse discrimination between convergence and divergence provided by the metric model, the partial order model allows different levels between full convergence (a total term as result) and full divergence ($\bot$ as result). Moreover, due to the equivalence to Böhm-reachability, we additionally obtain infinitary normalisation and infinitary confluence for orthogonal systems, which we do not have in the metric model, while still maintaining the compression property. While achieving the same goals as Böhm-extensions, the partial order approach provides an intuitive and more elegant model.

We have only studied strong convergence in this paper. It would be interesting to find out whether the shift to the partial order model has similar benefits for weak convergence, which is known to be rather unruly [Sim04].

Another interesting direction to follow is the ability to finitely simulate transfinite reductions by term graph rewriting. For m-convergence this is possible, at least to some extent [Ken94]. However, we think that a different approach to term graph rewriting, viz. the *double-pushout approach* [Ehr73] or the *equational approach* [Ari96], is more appropriate for p-convergence [Cor97, Bah09].

Acknowledgements

I want to thank Bernhard Gramlich for his constant support during the work on my master's thesis which made this work possible. I am also grateful for the valuable comments of the anonymous referees.

References

[Ari96] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *Fundam. Inf.*, 26(3-4):207–240, 1996.

[Arn80] André Arnold and Maurice Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundam. Inf.*, 3(4):445–476, 1980.

[Bah09] Patrick Bahr. *Infinitary Rewriting - Theory and Applications*. Master's thesis, Vienna University of Technology, Vienna, 2009.
URL `http://www.pa-ba.info/?q=pub/master`

[Blo04] Stefan Blom. An approximation based approach to infinitary lambda calculi. In Vincent van Oostrom (ed.), *RTA '04*, *Lecture Notes in Computer Science*, vol. 3091, pp. 221–232. Springer Berlin / Heidelberg, 2004. doi:10.1007/b98160.
URL `http://www.springerlink.com/content/4n3gqw43d1bpnldy/`

[Cor93] Andrea Corradini. Term rewriting in CT_Σ. In Marie-Claude Gaudel and Jean-Pierre Jouannaud (eds.), *TAPSOFT '93*, *Lecture Notes in Computer Science*, vol. 668, pp. 468–484. Springer Berlin / Heidelberg, 1993. doi:10.1007/3-540-56610-4_83.
URL `http://www.springerlink.com/content/f73r5p2v370220m4/`

[Cor97] Andrea Corradini and Frank Drewes. (Cyclic) term graph rewriting is adequate for rational parallel term rewriting. Tech. Rep. TR-14-97, Universita di Pisa, Dipartimento di Informatica, 1997.

[Der91] Nachum Dershowitz, Stéphane Kaplan, and David A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite, ... *Theor. Comput. Sci.*, 83(1):71–96, 1991. doi:DOI:10.1016/0304-3975(91)90040-9.
URL `http://www.sciencedirect.com/science/article/B6V1G-45DHJRB-H/2/767b35171dafdfa511dd0463ea25dbdd`

[Ehr73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *SWAT '73*, pp. 167–180. IEEE Computer Society, Washington, DC, USA, 1973. doi:http://dx.doi.org/10.1109/SWAT.1973.11.

[Kah93] Gilles Kahn and Gordon D. Plotkin. Concrete domains. *Theor. Comput. Sci.*, 121(1-2):187–277, 1993. doi:DOI:10.1016/0304-3975(93)90090-G.
URL `http://www.sciencedirect.com/science/article/B6V1G-45FC431-2K/2/6c30777ef97aea14c529418b4d5c5d4a`

[Kel55] John L. Kelley. *General Topology*, *Graduate Texts in Mathematics*, vol. 27. Springer-Verlag, 1955.

[Ken94] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Trans. Program. Lang. Syst.*, 16(3):493–523, 1994. doi:http://doi.acm.org/10.1145/177492.177577.

[Ken95] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Inform. and Comput.*, 119(1):18–38, 1995. doi:DOI:10.1006/inco.1995.1075.
URL `http://www.sciencedirect.com/science/article/B6WGK-45NJJYB-4W/2/7d48d04a2fe97d6e9e1fc5179f31a488`

[Ken99] Richard Kennaway, Vincent van Oostrom, and Fer-Jan de Vries. Meaningless terms in rewriting. *J. Funct. Logic Programming*, 1999(1):1–35, 1999.

[Ken03] Richard Kennaway and Fer-Jan de Vries. Infinitary rewriting. In Terese [Ter03], chap. 12, pp. 668–711.
URL `http://amazon.com/o/ASIN/0521391156/`

[Ket06] Jeroen Ketema. *Böhm-Like Trees for Rewriting*. Ph.D. thesis, Vrije Universiteit Amsterdam, 2006.
URL `http://dare.ubvu.vu.nl/handle/1871/9203`

[Rod98] Pieter Hendrik Rodenburg. Termination and confluence in infinitary term rewriting. *J. Symbolic Logic*, 63(4):1286–1296, 1998.
URL `http://www.jstor.org/stable/2586651`

[Sim04] Jakob Grue Simonsen. On confluence and residuals in Cauchy convergent transfinite rewriting. *Inf. Process. Lett.*, 91(3):141–146, 2004. doi:DOI:10.1016/j.ipl.2004.03.018.
URL `http://www.sciencedirect.com/science/article/B6V0F-4CBVNG4-1/2/d5d0f374f89fd62e07d023512a5b3dfe`

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edn., 2003.
URL `http://amazon.com/o/ASIN/0521391156/`

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 85-102
http://rewriting.loria.fr/rta/

UNIQUE NORMAL FORMS IN INFINITARY WEAKLY ORTHOGONAL TERM REWRITING

JÖRG ENDRULLIS [1] AND CLEMENS GRABMAYER [2] AND DIMITRI HENDRIKS [1] AND JAN WILLEM KLOP [1] AND VINCENT VAN OOSTROM [2]

[1] VU University Amsterdam, Dept. of Computer Science, de Boelelaan 1081a, 1081 HV Amsterdam

[2] Universiteit Utrecht, Department of Philosophy, Heidelberglaan 6, 3854 CS Utrecht
E-mail address, J. Endrullis: `joerg@few.vu.nl`

E-mail address, C. Grabmayer: `clemens@phil.uu.nl`

E-mail address, D. Hendriks: `diem@cs.vu.nl`

E-mail address, V. van Oostrom: `Vincent.vanOostrom@phil.uu.nl`

ABSTRACT. We present some contributions to the theory of infinitary rewriting for weakly orthogonal term rewrite systems, in which critical pairs may occur provided they are trivial.

We show that the infinitary unique normal form property (UN^∞) fails by a simple example of a weakly orthogonal TRS with two collapsing rules. By translating this example, we show that UN^∞ also fails for the infinitary $\lambda\beta\eta$-calculus.

As positive results we obtain the following: Infinitary confluence, and hence UN^∞, holds for weakly orthogonal TRSs that do not contain collapsing rules. To this end we refine the compression lemma. Furthermore, we consider the triangle and diamond properties for infinitary multi-steps (complete developments) in weakly orthogonal TRSs, by refining an earlier cluster-analysis for the finite case.

1. Introduction

While the theory of infinitary term rewriting is well-developed for orthogonal rewrite systems, much less is known about infinitary rewriting in non-orthogonal systems, in which critical pairs between rules may occur. In this paper we consider the simplest such systems, namely weakly orthogonal ones, in which all critical pairs are trivial. Conceptually, weakly orthogonal systems deviate little from orthogonal systems. But for the development of their rewrite theory specific notions and techniques had to be developed [5].

We show that the infinitary rewrite theory known for orthogonal systems fails dramatically in the case of weakly orthogonal systems. In Section 2, we give a simple counterexample

1998 ACM Subject Classification: D.1.1, D.3.1, F.4.1, F.4.2, I.1.1, I.1.3.

Key words and phrases: weakly orthogonal term rewrite systems, unique normal form property, infinitary rewriting, infinitary $\lambda\beta\eta$-calculus, collapsing rules, compression lemma.

© J. Endrullis, C. Grabmayer, D. Hendriks, J.W. Klop, and V. van Oostrom
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.85

to the infinitary unique normal form property UN^∞. Moreover, by a straightforward translation we obtain a counterexample to UN^∞ in the infinitary $\lambda\beta\eta$-calculus (Section 3), the paradigmatic example of a weakly orthogonal higher-order rewrite system.

In the remaining sections we show that, under simple restrictions, much of the theory of infinitary rewriting in orthogonal systems can be regained: we establish the diamond property, and consider the triangle property (Section 6) for weakly orthogonal TRSs without collapsing rules. An important ingredient in their proofs is a refinement of the compression lemma (Section 4).

For a general introduction to infinitary rewriting, as well as for notations used in this paper, we refer to [9, Ch.12], [6, 3].

2. A Counterexample to UN^∞ for Weakly Orthogonal Systems

In [3] it is shown that every orthogonal TRS exhibits the infinitary unique normal forms (UN^∞) property, see also [6]. In strong contrast, we will now give a counterexample showing that the UN^∞ property does *not* generalize to weakly orthogonal TRSs. The counterexample is very simple: its signature consists of the unary symbols P and S with the reduction rules: $\mathsf{P}(\mathsf{S}(x)) \to x$ and $\mathsf{S}(\mathsf{P}(x)) \to x$. Clearly this TRS is weakly orthogonal. In the sequel we consider the corresponding string rewrite system (SRS):

$$\mathsf{PS} \to \varepsilon \qquad\qquad \mathsf{SP} \to \varepsilon$$

where ε is the empty word. If w is a finite word, we write w^ω for the infinite word $www\ldots$. Using S and P we have infinite words such as $\zeta = (\mathsf{PS})^\omega$. Note that S^ω and P^ω are the only infinite normal forms, and that ζ only reduces to itself.

Given an infinite PS-word w we can plot in a graph the surplus number of S's of w when stepping through the word w from left to right, see e.g. Figure 1. The graph is obtained by counting S for $+1$ and P for -1. We define $\mathrm{sum}(w, n)$ as the result of this counting up to depth n in the word w (if w is finite we define $\mathrm{sum}(w) = \mathrm{sum}(w, |w|)$):

$$\mathrm{sum}(w, 0) = 0 \qquad \mathrm{sum}(\mathsf{S}w, n+1) = \mathrm{sum}(w, n) + 1 \qquad \mathrm{sum}(\mathsf{P}w, n+1) = \mathrm{sum}(w, n) - 1$$

For $w = (\mathsf{SP})^\omega$ the graph takes values, consecutively, $1, 0, 1, 0, \ldots$, for $w = \mathsf{S}^\omega$ it takes $1, 2, 3, \ldots$, and for $w = \mathsf{P}^\omega$ we have $-1, -2, -3, \ldots$.

We define the S-*norm* $\|w\|_\mathsf{S}$ and P-*norm* $\|w\|_\mathsf{P}$ of w:

$$\|w\|_\mathsf{S} = \sup_{n\in\mathbb{N}} \mathrm{sum}(w, n) \qquad\qquad \|w\|_\mathsf{P} = \sup_{n\in\mathbb{N}} (-\mathrm{sum}(w, n)) \tag{2.1}$$

So the S-norm (P-norm) of $(\mathsf{SP})^\omega$ is 1 (0), of S^ω it is ∞ (0), and of P^ω it is 0 (∞).

Lemma 2.1. *Let w be a finite PS-word, and let $n = \mathrm{sum}(w)$. If $n \geq 0$ then $w \twoheadrightarrow \mathsf{S}^n$, and $w \twoheadrightarrow \mathsf{P}^{-n}$, otherwise.*

Proof. For finite words u, v we have that $u \to v$ implies $\mathrm{sum}(u) = \mathrm{sum}(v)$. Moreover, $\to$ is normalising, and the only normal forms are of the form S^k and P^k for $k \geq 0$. ∎

Proposition 2.2.

(i) $w \twoheadrightarrow\!\!\!\!\rightarrow \mathsf{S}^\omega$ if and only if $\|w\|_\mathsf{S} = \infty$,
(ii) $w \twoheadrightarrow\!\!\!\!\rightarrow \mathsf{P}^\omega$ if and only if $\|w\|_\mathsf{P} = \infty$.

Proof. We consider only (i) as case (ii) can be treated analogously. From $\|w\|_\mathsf{S} = \infty$ it follows that $w = w_1 w_2 \ldots$ with finite words $w_1, w_2, \ldots$ such that $\mathrm{sum}(w_i) = 1$ for all $i \in \mathbb{N}$. Then $w_i \twoheadrightarrow \mathsf{S}$ for all $i \in \mathbb{N}$ by Lemma 2.1 and hence $w \twoheadrightarrow\!\!\!\!\rightarrow S^\omega$. ∎

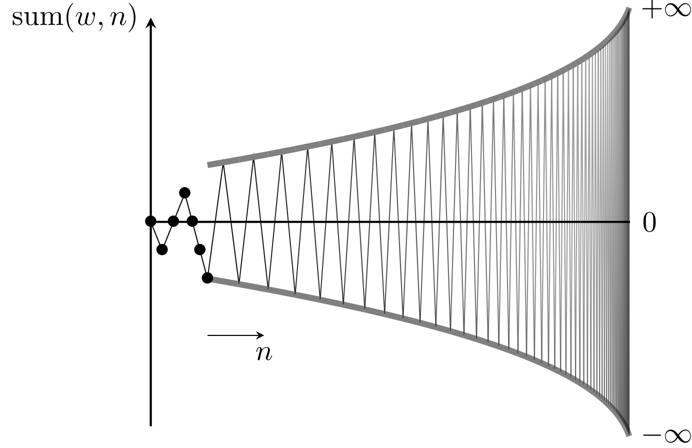


Figure 1: *Graph for the oscillating* PS*-word* $\psi = \mathsf{P}^1\,\mathsf{S}^2\,\mathsf{P}^3\,\ldots$.

Note that in Proposition 2.2 $w \twoheadrightarrow \mathsf{S}^\omega$ can always be achieved using the rule $\mathsf{PS} \to \varepsilon$ only. And likewise the rule $\mathsf{SP} \to \varepsilon$ for $w \twoheadrightarrow \mathsf{P}^\omega$.

Now let us take a term ψ with $\|\psi\|_\mathsf{S} = \infty$ *and* $\|\psi\|_\mathsf{P} = \infty$! Then by the previous proposition ψ reduces to both S^ω and P^ω, both normal forms. Hence UN^∞ fails. Indeed, such a term ψ can be found:

$$\psi = \mathsf{P\,SS\,PPP\,SSSS\,PPPPP\,SSSSSS}\,\ldots$$

The graph for this term is displayed in Figure 1. If we only apply rule $\mathsf{PS} \to \varepsilon$ the P-blocks are absorbed by the larger S-blocks to their right, leaving the normal form S^ω. Likewise, applying only $\mathsf{SP} \to \varepsilon$ yields P^ω.

We find that $\psi \twoheadrightarrow w$ for every infinite PS-word w, and more generally:

Proposition 2.3. *Every* PS*-word that reduces to both* S^ω *and* P^ω *reduces to any infinite* PS*-word.*

Proof. Let w be a PS-word such that $\mathsf{P}^\omega \twoheadleftarrow w \twoheadrightarrow \mathsf{S}^\omega$. And let u be the infinite PS-word we want to obtain. Then, by Proposition 2.2 we have that $\|w\|_\mathsf{P} = \|w\|_\mathsf{S} = \infty$. From this it follows that $w = w_1 w_2 \ldots$ with w_i finite PS-words such that $\mathrm{sum}(w_i) = 1$ if $u(i) = \mathsf{S}$ and $\mathrm{sum}(w_i) = -1$ if $u(i) = \mathsf{P}$. By Lemma 2.1, we get that $w_i \twoheadrightarrow u(i)$, and hence $w \twoheadrightarrow u$. ∎

Hence, not only is ψ a counterexample to UN^∞ for weakly orthogonal rewrite systems. But also, ψ rewrites to $(\mathsf{PS})^\omega$, a word which has no normal form. Thus, in contrast to orthogonal systems, weak normalisation is not preserved under infinite rewriting.

Figure 2 shows a more detailed analysis of various classes of PS-words. By Proposition 2.2 an infinite word w reduces to S^ω iff $\|w\|_\mathsf{S} = \infty$, and to P^ω iff $\|w\|_\mathsf{P} = \infty$. The shaded non-empty intersection ($\|w\|_\mathsf{S} = \|w\|_\mathsf{P} = \infty$) contains the counterexample term ψ mentioned above. All terms in this intersection are root-active (RA), that is, every $\twoheadrightarrow$-reduct can be reduced to a redex (at the root). However, there are also other root-active terms. For example $\xi = \mathsf{S\,P\,S^2\,P^2\,S^3\,P^3}\ldots$ is a root-active term which reduces to S^ω but not to P^ω (i.e., $\|\xi\|_\mathsf{P} = 0 < \infty$ and $\|\xi\|_\mathsf{S} = \infty$). The term $\xi' = \mathsf{S}\,\xi$ (a reduct of ξ) is not root-active but still not SN^∞, yet it reduces to S^ω. An example of a root-active term which reduces only to itself (implying that $\|\xi\|_\mathsf{S}$ and $\|\xi\|_\mathsf{P}$ are finite) is $\zeta = (\mathsf{P\,S})^\omega$. The dotted part consists of

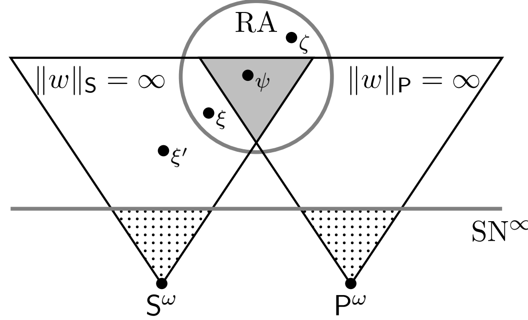


Figure 2: *Venn diagram of infinite* PS*-words.*

terms with the property of infinitary strong normalization (SN^{∞},[6]), normalizing to S^{ω}, or P^{ω}, respectively. For instance $(\mathsf{S}\,\mathsf{S}\,\mathsf{P})^{\omega}$ is in the left dotted triangle.

The root-active terms can be characterized as follows.

Proposition 2.4. *A* PS*-word w is root-active if and only if w is the concatenation of infinitely many finite 'zero-words' $w_1, w_2, w_3, \ldots$, that is, words w_i with* $\mathrm{sum}(w_i) = 0$*.* ∎

As a consequence of this proposition, an infinite PS-word w is root-active if and only if $\mathrm{sum}(w, n) = 0$ for infinitely many n, and hence, if $((\liminf)_{n\to\infty}|\mathrm{sum}(w, n)|) = 0$.

Corollary 2.5. *For an infinite* PS*-word w we have* $\mathrm{SN}^{\infty}(w)$ *if and only if each value* $\mathrm{sum}(w, n)$ *for $n = 0, 1\ldots$ occurs only finitely often.* ∎

It follows that $\mathrm{SN}^{\infty}(w)$ holds if and only if $((\liminf)_{n\to\infty}|\mathrm{sum}(w, n)|) = \infty$, and hence, if $\lim_{n\to\infty} \mathrm{sum}(w, n) \in \{\infty, -\infty\}$.

3. A Counterexample to UN^{∞} of the Infinitary $\lambda\beta\eta$-Calculus

We give a straightforward translation of the word $\psi = \mathsf{P}^1\,\mathsf{S}^2\,\mathsf{P}^3\,\ldots$ from the previous section into an infinite λ-term which then forms a counterexample to the infinitary unique normal form property UN^{∞} for $\lambda^{\infty}\beta\eta$, the infinitary $\lambda\beta\eta$-calculus. The infinitary $\lambda\beta\eta$-calculus [7, 8] is a well-known example of a weakly orthogonal higher-order term rewrite system.

The set $\mathit{Ter}^{\infty}(\lambda)$ of (potentially) infinite λ-terms is coinductively defined by:

$$M ::= x \mid MM \mid \lambda x.M \qquad (\mathit{Ter}^{\infty}(\lambda))$$

The rewrite rules of $\lambda^{\infty}\beta\eta$ are:

$$\lambda x.MN \to M[x{:=}N] \qquad (\beta)$$

$$\lambda x.Mx \to M \qquad \text{if } x \text{ is not free in } M \qquad (\eta)$$

where $M[x{:=}N]$ denotes the result of substituting N for all free occurrences of x in M. The $\lambda^\infty\beta\eta$-calculus allows for two critical pairs[1]:

$$Mx \stackrel{\beta}{\leftarrow} (\lambda x.Mx)x \stackrel{\eta}{\rightarrow} Mx \qquad\qquad \lambda x.M[y{:=}x] \stackrel{\beta}{\leftarrow} \lambda x.(\lambda y.M)x \stackrel{\eta}{\rightarrow} \lambda y.M$$

As we have that $\lambda x.M[y{:=}x]$ and $\lambda y.M$ are equal modulo renaming of bound variables, both of these critical pairs are trivial. Hence $\lambda^\infty\beta\eta$ is weakly orthogonal.

We translate infinite PS-words to λ-terms.

[1]We use the notation of infinitary λ-calculus, but we view the rule schemes (β) and (η) as rules of a second-order HRS, thereby obtaining a formal notion of critical pairs ([9, Def. 11.6.10]). Likewise, CRSs can be viewed as second-order HRSs.

Definition 3.1. We define $(\!|_|\!) : \{\mathsf{P},\mathsf{S}\}^\omega \to \mathit{Ter}^\infty(\lambda)$ by $(\!|w|\!) = (\!|w|\!)_0$, for all $w \in \{\mathsf{P},\mathsf{S}\}^\omega$, where $(\!|w|\!)_i$ is defined coinductively, for all $i \in \mathbb{Z}$, as follows:

$$(\!|\mathsf{P}w|\!)_i = (\!|w|\!)_{i-1}\, x_i \qquad\qquad (\!|\mathsf{S}w|\!)_i = \lambda x_{i+1}.(\!|w|\!)_{i+1}$$

The translation of ψ is the λ-term $(\!|\psi|\!)$, displayed in the middle of Figure 3. This term has

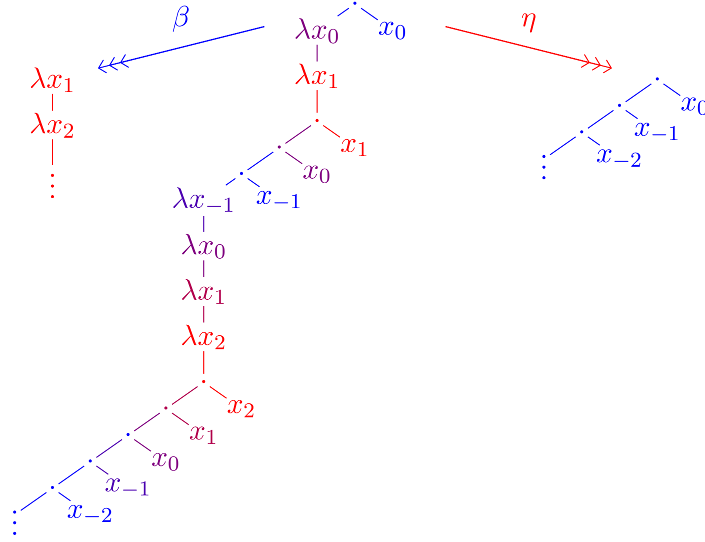


Figure 3: *Counterexample to unique normal forms in $\lambda^\infty\beta\eta$.*

two normal forms (corresponding to S^ω and P^ω), as indicated in the figure.

While $(\!|\psi|\!)$ cannot be generated from a finite λ-term (it has infinitely many free variables), the finite term WWI where $W = \lambda wf.f(ww(\lambda abc.f(abc))x_0)$ and $I = \lambda a.a$ exhibits a similar behaviour, reducing both to $A = \lambda x.A$ and $B = Bx_0$. This can be seen as follows: Let $V_n = \lambda v_1 \ldots v_n.(v_1 \ldots v_n)$. First note that $WWI \to_\beta^2 I(WW(\lambda abc.I(abc))x_0) \to_\beta^2 WWV_3x_0$. Then we get:

$$\begin{aligned} WWV_3x_0 &\to_\beta^2 V_3(WW(\lambda abc.V_3(abc))x_0)x_0 \to_\beta^3 \lambda v_3.WWV_5x_0x_0v_3 \\ &\to_\beta^6 \lambda v_3v_5.WWV_7x_0x_0x_0v_3v_5 \twoheadrightarrow_\beta \lambda v_3v_5v_7 \ldots. =_\alpha A \\ WWV_3x_0 &\to_\eta^2 (WWI)x_0 \twoheadrightarrow_{\beta\eta} B \end{aligned}$$

Note that the number of bound variables needed along the reduction from $WW(\lambda a.a)$ to A is unbounded, but that A can be written using only a single one. We conjecture that it holds for every counterexample to UN^∞ in the infinitary $\lambda\beta\eta$-calculus that during the rewrite process to one of the normal forms unboundedly many variables are needed.

The translation given in Definition 3.1 lifts $\mathsf{PS} \to \varepsilon$ to β, and $\mathsf{SP} \to \varepsilon$ to η.

Lemma 3.2. *An application of the rule $\mathsf{PS} \to \varepsilon$ at depth k in an infinite PS-word w corresponds to a β-step in $\lambda^\infty\beta\eta$ at depth k in $(\!|w|\!)_i$. Similarly so for the rule $\mathsf{SP} \to \varepsilon$ and the η-rule. These correspondences are indicated in the following diagrams:*

$$\begin{array}{ccc} \mathsf{PS}w & \xrightarrow{(\!|_|\!)_i} & (\lambda x_i.(\!|w|\!)_i)\,x_i \\ \mathsf{PS}\downarrow & & \downarrow\beta \\ w & \xrightarrow{(\!|_|\!)_i} & (\!|w|\!)_i \end{array} \qquad \begin{array}{ccc} \mathsf{SP}w & \xrightarrow{(\!|_|\!)_i} & \lambda x_{i+1}.(\!|w|\!)_i\,x_{i+1} \\ \mathsf{SP}\downarrow & & \downarrow\eta \\ w & \xrightarrow{(\!|_|\!)_i} & (\!|w|\!)_i \end{array}$$

The counterexample to the infinitary unique normal form property UN^∞ for infinitary $\lambda\beta\eta$-calculus ($\lambda^\infty\beta\eta$) establishes a striking contrast to the situation for infinitary $\lambda\beta$-calculus ($\lambda^\infty\beta$). In the latter, infinitary confluence breaks down, but infinitary normal forms stay unique. Therefore $\lambda^\infty\beta$ clearly is of importance in the model theory of λ-calculus; for several models the equality is captured by convertibility in $\lambda^\infty\beta$. E.g. Böhm Trees, Lévy–Longo trees and Berarducci trees are unique normal forms in this rewrite system, when suitable $\bot$-normalization rules are added. (See [1, 2] and [9, Ch.12]). However, when the η-rule is added, and the infinitary perspective is maintained, then 'everything' breaks down dramatically: not only infinitary confluence, but also unique infinitary normal forms.

From the perspective of combinatory reduction systems (CRSs, see [9]) the η-rule has many undesirable properties: (i) it is undecidable whether an infinite term is an η-redex, since it is undecidable whether an infinite term contains a variable freely; (ii) single-step η-reduction is not lower semi-continuous: if t η-reduces to u, then for a given $\epsilon > 0$ we cannot always find a $\delta > 0$ such that anything within δ-distance of t η-reduces to something within ϵ-distance of u; (iii) the η-rule is not fully-extended, and various existing results for orthogonal infinite CRSs require fully-extendedness, see [4].

4. A Refinement of the Compression Lemma

As a preparation for Section 5 we will prove the following lemma, which is a refined version of the Compression Lemma in left-linear TRSs. In its original formulation (e.g. see Theorem 12.7.1 on page 689 in [9]), it states that strongly convergent rewrite sequences in left-linear TRSs can be compressed to length less or equal to ω. We recall that a rewrite sequence of ordinal length α is strongly convergent if for each limit ordinal $\lambda \leq \alpha$ the depth of the contracted redexes tends to infinity.

Lemma 4.1 (Refined Compression Lemma)**.** *Let R be a left-linear iTRS. Let $\kappa : s \to_R^\alpha t$ be a rewrite sequence, d the minimal depth of a step in κ, and n the number of steps at depth d in κ. Then there exists a rewrite sequence $\kappa' : s \to_R^{\leq\omega} t$ in which all steps take place at depth $\geq d$, and where precisely n steps contract redexes at depth d.*

Proof. We proceed by transfinite induction on the ordinal length α of rewrite sequences $\kappa : s \to_R^\alpha t$ with d the minimal depth of a step in κ, and n the number of steps at depth d in κ.

In case that $\alpha = 0$ nothing needs to be shown.

Suppose α is a successor ordinal. Then $\alpha = \beta + 1$ for some ordinal β, and κ is of the form $s \to^\beta s' \to t$. Applying the induction hypothesis to $s \to^\beta s'$ yields a rewrite sequence $s \to^\gamma s'$ of length $\gamma \leq \omega$ that contains the same number of steps at depth d, and no steps at depth less than d.

If $\gamma < \omega$, then $s \to^\gamma s' \to t$ is a rewrite sequence of length $\gamma + 1 < \omega$, in which all steps take place at depth $\geq d$ and precisely n steps at depth d.

If $\gamma = \omega$, we obtain a rewrite sequence of the form $s \equiv s_0 \to s_1 \to \ldots \to^\omega s_\omega \to t$. Let $\ell \to r \in R$ be the rule applied in the final step $s_\omega \to t$, that is, $s_\omega \equiv C[\ell\sigma] \to C[r\sigma] \equiv t$ for

some context C and substitution σ. Moreover, let d_h be the depth of the hole in C, and d_p the depth of the pattern of ℓ. Since the reduction $s_0 \twoheadrightarrow^{\omega} s_\omega$ is strongly convergent, there exists $n \in \mathbb{N}$ such that all rewrite steps in $\xi : s_n \twoheadrightarrow^{\omega} s_\omega$ have depth $> d_h + d_p$, and hence are below the pattern of the redex contracted in the last step $s_\omega \to t$. As a consequence, there exists a context D and a substitution τ such that $s_n \equiv D[\ell\tau]$. Since the rewrite sequence $\xi : s_n \equiv D[\ell\tau] \twoheadrightarrow^{\omega} C[\ell\sigma] \equiv s_\omega$ consists only of steps at depth $> d_h + d_p$, it follows that:

- there exists a rewrite sequence $\vartheta : D[\Box] \twoheadrightarrow^{\leq\omega} C[\Box]$ at depth $> d_h + d_p$, and
- there exist rewrite sequences $\vartheta_x : \tau(x) \twoheadrightarrow^{\leq\omega} \sigma(x)$ for all $x \in \mathit{Var}(\ell)$.

We now prepend the final step $s_\omega \to t$ to s_n, that is: $s_n \equiv D[\ell\tau] \to D[r\tau]$. Even if the term r is infinite, this creates at most ω-many copies of subterms $\tau(x)$ with reduction sequences $\vartheta_x : \tau(x) \twoheadrightarrow^{\leq\omega} \sigma(x)$ of length $\leq \omega$. Since the rewrite sequences ϑ and ϑ_x for $x \in \mathit{Var}(\ell)$ are in disjoint (parallel) subterms, there exists an interleaving $D[r\tau] \twoheadrightarrow^{\leq\omega} C[r\sigma]$ of length at most ω (the idea is similar to establishing countability of ω^2 by dovetailing). We obtain a rewrite sequence $\kappa' : s \twoheadrightarrow^{\leq\omega} t$, since $s \to^n s_n \equiv D[\ell\tau] \to D[r\tau] \twoheadrightarrow^{\leq\omega} C[r\sigma] \equiv t$.

It remains to be shown that κ' contains only steps at depth $\geq d$, and that it has the same number of steps as the original sequence κ at depth d. This follows from the induction hypothesis and the fact that all steps in $s_n \twoheadrightarrow^{\omega} s_\omega$ have depth $> d_h + d_p$ and thus also all steps of the interleaving $D[r\tau] \twoheadrightarrow^{\leq\omega} C[r\sigma]$ have depth $> d_h + d_p - d_p = d_h \geq d$ (the application of $\ell \to r$ can lift steps at most by the pattern depth d_p of ℓ).

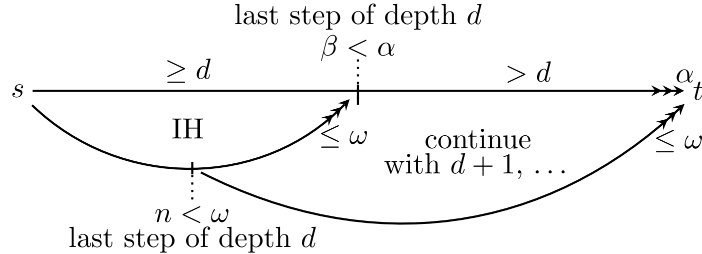


Figure 4: *Compression Lemma, in case α is a limit ordinal.*

Finally, suppose that α is a limit ordinal $> \omega$. We refer to Figure 4 for a sketch of the proof. Since κ is strongly convergent, only a finite number of steps take place at depth d. Hence there exists $\beta < \alpha$ such that s_β is the target of the last step at depth d in κ. We have $s \twoheadrightarrow^{\beta} s_\beta \twoheadrightarrow^{\leq\alpha} t$ and all rewrite steps in $s_\beta \twoheadrightarrow^{\leq\alpha} t$ are at depth $> d$. By induction hypothesis there exists a rewrite sequence $\xi : s \twoheadrightarrow^{\leq\omega} s_\beta$ containing an equal amount of steps at depth d as $s \twoheadrightarrow^{\beta} s_\beta$. Consider the last step of depth d in ξ . This step has a finite index $n < \omega$. Thus we have $s \to^* s_n \twoheadrightarrow^{\leq\alpha} t$, and all steps in $s_n \twoheadrightarrow^{\leq\alpha} t$ are at depth $> d$. By successively applying this argument to $s_n \twoheadrightarrow^{\leq\alpha} t$ we construct finite initial segments $s \to^* s_n$ with strictly increasing minimal rewrite depth d. Concatenating these finite initial segments yields a reduction $s \twoheadrightarrow^{\leq\omega} t$ containing as many steps at depth d as the original sequence. ∎

With this refined compression lemma we now prove that also divergent rewrite sequences can be compressed to length less or equal to ω.

Theorem 4.2. *Let R be a left-linear iTRS. For every divergent rewrite sequence $\kappa : s \to_R^{\alpha}$ of length α there exists a divergent rewrite sequence $\kappa' : s \to_R^{\leq\omega}$ of length less or equal to ω.*

Proof. Let $\kappa : s \to_R^\alpha$ be a divergent rewrite sequence. Then there exist $k \in \mathbb{N}$ such that infinitely many steps in κ take place at depth k. Let d be the minimum of all numbers k with that property. Let β be the index of the last step above depth d in κ, $\kappa : s \to^\beta s_\beta \to^{\leq\alpha}$. Then by Lemma 4.1 the rewrite sequence $s \to^\beta s_\beta$ can be compressed to a rewrite sequence $s \to^{\leq\omega} s_\beta$ such that $s_\beta \to^{\leq\alpha}$ consists only of steps at depth $\geq d$, among which infinitely many steps are at depth d. Let n be the index of the last step of depth $\leq d$ in the rewrite sequence $s \to^{\leq\omega} s_\beta$. Then $s \to^* s_n \to^{\leq\omega} s_\beta \to^{\leq\alpha}$, and $s_n \to^{\leq\omega} s_\beta \to^{\leq\alpha}$ contains only steps at depth $\geq d$. Thus all steps with depth less than d occur in the finite prefix $s \to^* s_n$.

Now consider the rewrite sequence $\kappa_1 : s_n \to^{\leq\omega} \cdot \to^{\leq\alpha}$, say $\kappa_1 : s_n \to^\gamma$ for short, containing infinitely many steps at depth d. Let γ' be the index of the first step at depth d in κ_1. Then $\kappa_1 : s_n \to^{\gamma'} u \to^{\leq\gamma}$ for some term u and $s_n \to^{\gamma'} u$ can be compressed to $s_n \to^{\leq\omega} u$ containing exactly one step at depth d. Now let m be the index of this step, then $s_n \to^m u' \to^{\leq\omega} u \to^{\leq\gamma}$ where $s_n \to^m u'$ contains one step at depth d. Repeatedly applying this construction to $u' \to^{\leq\omega} u \to^{\leq\gamma}$ we obtain a rewrite sequence $\kappa' : s \to^* s_n \to^* u' \to^* u'' \to \ldots$ that contains infinitely many steps at depth d, and hence is divergent. ∎

5. Infinitary Confluence

In Section 2 we have seen that the property UN^∞ fails for weakly orthogonal TRSs when collapsing rules are present, and hence also CR^∞. Now we show that weakly orthogonal TRSs without collapsing rules are infinitary confluent (CR^∞), and as a consequence also have the property UN^∞.

We adapt the projection of parallel steps in weakly orthogonal TRSs from [9, Section 8.8.4.] to infinite terms. The basic idea is to orthogonalize the parallel steps, and then project the orthogonalized steps. The orthogonalization uses that overlapping redexes have the same effect and hence can be replaced by each other. In case of overlaps we replace the outermost redex by the innermost one. This is possible since the maximal nesting depth of the union of two infinite parallel steps is at most 2, that is, there can not be infinite chains of overlapping nested redexes in such a union (see Example 6.3). For a treatment of infinitary multi-steps where such chains can occur, we refer to Section 6. See further [9, Proposition 8.8.23] for orthogonalization in the finitary case.

Definition 5.1. Let R be a TRS, and $t \in \mathit{Ter}^\infty(\Sigma)$ a term.

A *redex* in t is a pair consisting of a position p and a rule $\ell \to r$, such that $t|_p = \ell^\sigma$ for some substitution σ. We call p and $\ell \to r$ the *root* and *rule* of the redex, respectively. The pattern of a redex $\langle p, \ell \to r\rangle$ is the set of all positions pq such that $\ell(q)$ is a function symbol.

Two sets of positions are *overlapping* if they have a non-empty intersection. For redexes u and v in t we say that *u and v overlap*, denoted by $u \leftrightsquigarrow v$, if the patterns of u and v overlap. A set U of redexes is called *non-overlapping* if, for all $u, v \in U$ with $u \neq v$, u does not overlap with v.

For a study of developments we refer to [9, Sec. 4.5.2] and [10]. Here, we briefly introduce developments and multi-steps via labelling (underlining).

Definition 5.2. Let R be a weakly orthogonal TRS over Σ. For symbols $f \in \Sigma$ and $\rho \in R$ we write f^ρ for f labelled with ρ. For labelled terms t, we write $\lfloor t \rfloor$ to denote the term obtained from t by dropping all labels.

We define the TRS $R^\triangleright$ to consist of all rules $\ell^\rho \to r$ for $\rho : \ell \to r \in R$ where ℓ^ρ is the

term obtained from ℓ by labelling the root-symbol of ℓ with ρ.

Let $t, t' \in \mathit{Ter}^{\infty}(\Sigma)$ be terms, and U a set of non-overlapping redexes in t. Let t^U be the term obtained from t by labelling for each redex $\langle p, \rho \rangle \in U$ the symbol at position p in t with ρ. A *development of* U in t is a rewrite sequence $t \twoheadrightarrow_R t'$ (in R) that can be lifted to a reduction $t^U \twoheadrightarrow_{R^\triangleright} t''$ (in $R^\triangleright$) such that t'' arises from t' by adding some labels; the development is called *complete* if $t' \equiv t''$. A *multi-step with respect to* U is a step $t \multimap_U t'$ such that there exists a reduction $t^U \twoheadrightarrow_{R^\triangleright} t'$.

In non-collapsing, weakly orthogonal TRSs, every set U of non-overlapping redexes has a complete development, and every complete development of U ends in the same term [9]. Multi-steps arise from complete developments, and are uniquely determined by their starting term and redex set.

Definition 5.3. Let R be a TRS, $t \in \mathit{Ter}^{\infty}(\Sigma)$ a term, and let U and V be sets of redexes in t. We call U and V *orthogonal (to each other)* if $U \cup V$ is a non-overlapping set of redexes.

Definition 5.4. Let R be a non-collapsing, weakly orthogonal TRS, and let U and V be orthogonal sets of redexes in a term t. For multi-steps $\phi : t \multimap_U t'$ and $\psi : t \multimap_V t''$ with respect to U and V we define the projection ϕ/ψ as the multi-step $t'' \multimap_{U'} s$ with respect to the set of residuals $U' = U/\psi$ as defined in [9].[1] In the sequel we frequently write $\multimap$ for the multi-step relation, suppressing the set of redexes U that induces the multi-step $\multimap_U$.

Definition 5.5. An *orthogonalization* of a pair $\langle \phi, \psi \rangle$ of multi-steps $\phi : s \multimap_U t_1$ and $\psi : s \multimap_V t_2$ with respect to sets U and V of redexes in s is a pair $\langle \phi', \psi' \rangle$ of multi-steps $\phi' : s \multimap_{U'} t_1$ and $\psi' : s \multimap_{V'} t_2$ with respect to orthogonal sets U' and V' of redexes in s.

A parallel step $\phi : s \mathrel{+\!\!\!\!\rightarrow} t$ is a multi-step $\phi : s \multimap_U t$ with respect to a set U of parallel redexes, that is, redexes at pairwise disjoint positions.

Proposition 5.6. *Let $\phi : s \mathrel{+\!\!\!\!\rightarrow} t_1$ and $\psi : s \mathrel{+\!\!\!\!\rightarrow} t_2$ be parallel steps in a weakly orthogonal TRS. Then there exists an orthogonalization $\langle \phi', \psi' \rangle$ of ϕ and ψ with the special property that $\phi' : s \mathrel{+\!\!\!\!\rightarrow} t_1$ and $\psi' : s \mathrel{+\!\!\!\!\rightarrow} t_2$.*

Proof. In case of overlaps between U and V, then for every overlap we replace the outermost redex by the innermost one (if there are multiple inner redexes overlapping, then we choose the left-most among the top-most redexes). If there are two redexes at the same position but with respect to different rules, then we replace the redex in V with the one in U. See also Figure 5. ∎

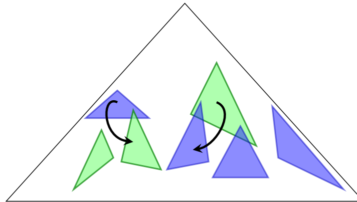


Figure 5: *Orthogonalization of parallel steps; the arrow indicates replacement.*

Definition 5.7. Let $\phi : s \mathrel{+\!\!\!\!\rightarrow} t_1$, $\psi : s \mathrel{+\!\!\!\!\rightarrow} t_2$ be parallel steps in a weakly orthogonal TRS. The *weakly orthogonal projection ϕ/ψ of ϕ over ψ* is defined as the orthogonal projection

[1] We refer to Def. 12.5.3 in [9], and note that the definition not only applies in orthogonal TRSs, but also to every non-overlapping set U of redexes versus a multistep ϕ w.r.t. a redex set V that is orthogonal to U.

ϕ'/ψ' where $\langle\phi', \psi'\rangle$ is the orthogonalization of ϕ and ψ given in the proof of Proposition 5.6.

Remark 5.8. The weakly orthogonal projection does not give rise to a residual system in the sense of [9]. The projection fulfils the three identities $\phi/\phi \approx 1$, $\phi/1 \approx \phi$, and $1/\phi \approx 1$, but not the *cube identity* $(\phi/\psi)/(\chi/\psi) \approx (\phi/\chi)/(\psi/\chi)$.

Lemma 5.9. *Let $\phi : s \twoheadrightarrow t_1$, $\psi : s \twoheadrightarrow t_2$ be parallel steps in a weakly orthogonal TRS R. Let d_ϕ and d_ψ be the minimal depth of a step in ϕ and ψ, respectively. Then the minimal depth of the weakly orthogonal projections ϕ/ψ and ψ/ϕ is greater or equal* $\min(d_\phi, d_\psi)$*. If R contains no collapsing rules then the minimal depth of ϕ/ψ and ψ/ϕ is greater or equal* $\min(d_\phi, d_\psi + 1)$ *and* $\min(d_\psi, d_\phi + 1)$*, respectively.*

Proof. Immediate from the definition of the orthogonalization (for overlaps the innermost redex is chosen) and the fact that in the orthogonal projection a non-collapsing rule applied at depth d can lift nested redexes at most to depth $d + 1$ (but not above). ∎

Lemma 5.10 (Parallel Moves Lemma)**.** *Let R be a weakly orthogonal TRS, $\kappa : s \to^\alpha t_1$ a rewrite sequence, and $\phi : s \twoheadrightarrow t_2$ a parallel rewrite step. Let d_κ and d_ϕ be the minimal depth of a step in κ and ϕ, respectively. Then there exist a term u, a rewrite sequence $\xi : t_2 \to^{\leq\omega} u$ and a parallel step $\psi : t_1 \twoheadrightarrow u$ such that the minimal depth of the rewrite steps in ξ and ψ is* $\min(d_\kappa, d_\xi)$*; see Figure 6 (left).*

If additionally R contains no collapsing rules, then the minimal depth of a step in ξ and ψ is $\min(d_\kappa, d_\xi + 1)$ *and* $\min(d_\xi, d_\kappa + 1)$*, respectively. See also Figure 6 (right).*

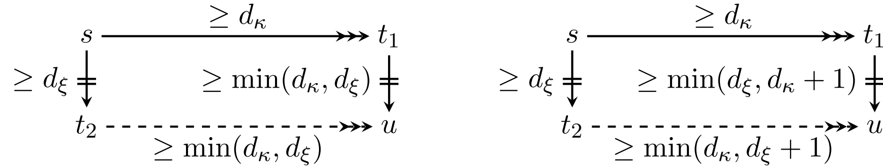


Figure 6: *Parallel Moves Lemma; with (left) and without (right) collapsing rules.*

Proof. By compression we may assume $\alpha \leq \omega$ in $\kappa : s \to^{\leq\omega} t_1$ (note that, the minimal depth d is preserved by compression). Let $\kappa : s \equiv s_0 \to s_1 \to s_2 \to \ldots$, and define $\psi_0 = \psi$. Furthermore, let $\kappa_{\leq n}$ denote the prefix of κ of length n, that is, $s_0 \to \ldots \to s_n$ and let $\kappa_{\geq n}$ denote the suffix $s_n \to s_{n+1} \to \ldots$ of κ. We employ the projection of parallel steps to close the elementary diagrams with top $s_n \to s_{n+1}$ and left $\psi_n : s_n \twoheadrightarrow s'_n$, that is, we construct the projections $\psi_{i+1} = \psi_i/(s_i \to s_{i+1})$ (right) and $(s_i \to s_{i+1})/\psi_i$ (bottom). Then by induction on n using Lemma 5.9 there exists for every $1 \leq n \leq \alpha$ a term s'_n, and parallel steps $\phi_n : s_n \twoheadrightarrow s'_n$ and $s'_{n-1} \twoheadrightarrow s'_n$. See Figure 7 for an overview.

We show that the rewrite sequence constructed at the bottom $s'_0 \twoheadrightarrow s'_1 \twoheadrightarrow \ldots$ of Figure 7 is strongly convergent, and that the parallel steps ϕ_i have a limit for $i \to \infty$ (parallel steps are always strongly convergent).

Let $d \in \mathbb{N}$ be arbitrary. By strong convergence of κ there exists $n_0 \in \mathbb{N}$ such that all steps in $\kappa_{\geq n_0}$ are at depth $\geq d$. Since ϕ_{n_0} is a parallel step there are only finitely many redexes $\phi_{n_0,<d} \subseteq \phi_{n_0}$ in ϕ_{n_0} rooted above depth d. By projection of ϕ_{n_0} along $\kappa_{\geq n_0}$ no fresh redexes above depth d can be created. The steps in $\phi_{n_0,<d}$ may be cancelled out due to overlaps, nevertheless, for all $m \geq n_0$ the set of steps above depth d in ϕ_m is a subset of $\phi_{n_0,<d}$.

Let p be the maximal depth of a left-hand side of a rule applied in $\phi_{n_0,<d}$. By strong

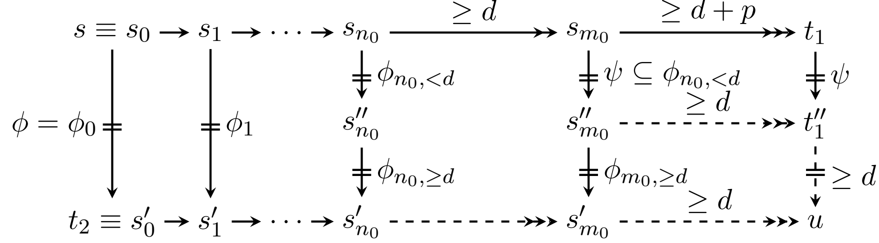


Figure 7: *Parallel Moves Lemma, proof overview.*

convergence of κ there exists $m_0 \geq n_0 \in \mathbb{N}$ such that all steps in $\kappa_{\geq n_0}$ are at depth $\geq d+p$. As a consequence the steps ψ in ϕ_{m_0} rooted above depth d will stay fixed throughout the remainder of the projection. Then for all $m \geq m_0$ the parallel step ϕ_m can be split into $\phi_m = s_m \twoheadrightarrow_\psi s''_m \twoheadrightarrow_{\phi_{m,\geq d}} s'_m$ where $\phi_{m,\geq d}$ consists of the steps of ϕ_m at depth $\geq d$. Since d was arbitrary, it follows that projection of ϕ over κ has a limit. Moreover the steps of the projection of $\kappa_{\geq m_0}$ over ϕ_{m_0} are at depth $\geq d+p-p = d$ since rules with pattern depth $\leq p$ can lift steps by at most by p. Again, since d was arbitrary, it follows that the projection of κ over ϕ is strongly convergent.

Finally, both constructed rewrite sequences (bottom and right) converge towards the same limit u since all terms $\{s'_m, s''_m \mid m \geq m_0\}$ coincide up to depth $d-1$ (the terms $\{s_m \mid m \geq m_0\}$ coincide up to depth $d+p-1$ and the lifting effect of the steps ϕ_m is limited by p). ∎

Theorem 5.11. *Every weakly orthogonal TRS without collapsing rules is infinitary confluent.*

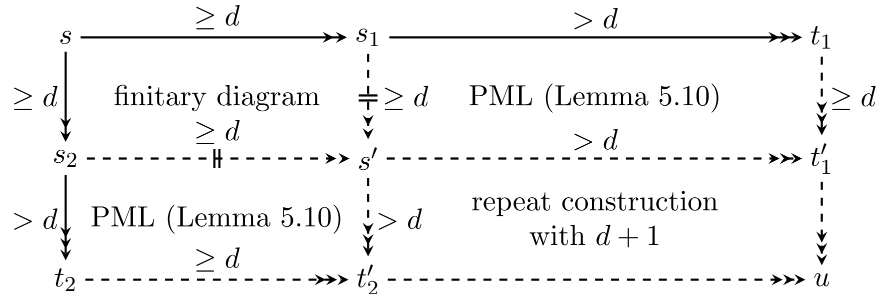


Figure 8: *Infinitary confluence.*

Proof. An overview of the proof is given in Figure 8. Let $\kappa : s \to^\alpha t_1$ and $\xi : s \to^\beta t_2$ be two rewrite sequences. By compression we may assume $\alpha \leq \omega$ and $\beta \leq \omega$. Let d be the minimal depth of any rewrite step in κ and ξ. Then κ and ξ are of the form $\kappa : s \to^* s_1 \to^{\leq\omega} t_1$ and $\xi : s \to^* s_2 \to^{\leq\omega} t_2$ such that all steps in $s_1 \to^{\leq\omega} t_1$ and $s_2 \to^{\leq\omega} t_2$ at depth $> d$.

Then $s \to^* s_1$ and $s \to^* s_2$ can be joined by finitary diagram completion employing the diamond property for parallel steps. If follows that there exists a term s' and finite sequences of (possibly infinite) parallel steps $s_1 \twoheadrightarrow^* s'$ and $s_2 \twoheadrightarrow^* s'$ all steps of which are at depth $\geq d$ (Lemma 5.9). We project $s_1 \to^{\leq\omega} t_1$ over $s_1 \twoheadrightarrow^* s'$, $s_2 \to^{\leq\omega} t_2$ over $s_2 \twoheadrightarrow^* s'$ by repeated application of the Lemma 5.10, obtaining rewrite sequences $t_1 \twoheadrightarrow t'_1$, $s' \twoheadrightarrow t'_1$, $t_2 \twoheadrightarrow t'_2$, and $s' \twoheadrightarrow t'_2$ with depth $\geq d$, $> d$, $\geq d$, and $> d$, respectively. As a consequence we

have t'_1, s' and t'_2 coincide up to (including) depth d. Recursively applying the construction to the rewrite sequences $s' \twoheadrightarrow t'_1$ and $s' \twoheadrightarrow t'_2$ yields strongly convergent rewrite sequences $t_2 \twoheadrightarrow t'_2 \twoheadrightarrow t''_2 \twoheadrightarrow \ldots$ and $t_1 \twoheadrightarrow t'_1 \twoheadrightarrow t''_1 \twoheadrightarrow \ldots$ where the terms $t_1^{(n)}$ and $t_2^{(n)}$ coincide up to depth $d + n - 1$. Thus these rewrite sequences converge towards the same limit u. ∎

We consider an example to illustrate that the absence of collapsing rules is a necessary condition for Theorem 5.11.

Example 5.12. Let R be a TRS over the signature $\{f, a, b\}$ consisting of the collapsing rule: $f(x, y) \to x$ Then, using a self-explaining recursive notation, the term $s = f(f(s, b), a)$ rewrites in ω many steps to $t_1 = f(t_1, a)$ as well as $t_2 = f(t_2, b)$ which have no common reduct. The TRS R is weakly orthogonal (even orthogonal) but not confluent. The same phenomenon occurs in the infinitary version of combinatory logic, due to the rule $Kxy \to x$.

6. The Diamond and Triangle Property for Multi-Steps

We prove that infinitary multi-steps in weakly orthogonal TRSs without collapsing rules have the diamond property. For all TRSs in this section we assume that are weakly orthogonal and do not contain collapsing rules.

Definition 6.1. A binary relation $\to$ on A is said to have:

- the *diamond property* if $\leftarrow \cdot \to \subseteq \to \cdot \leftarrow$, and
- the *triangle property* if $\forall a \in A.\, \exists a' \in A.\, a \to a' \wedge (\forall b \in A.\, a \to b \Rightarrow b \to a')$.

We develop an orthogonalization algorithm that, given two co-initial multisteps, makes them orthogonal to each other by eliminating overlaps. Since overlapping steps in weakly orthogonal TRSs have the same targets, we can replace one by the other. The challenge is to do this in such a way that no new overlaps are created.

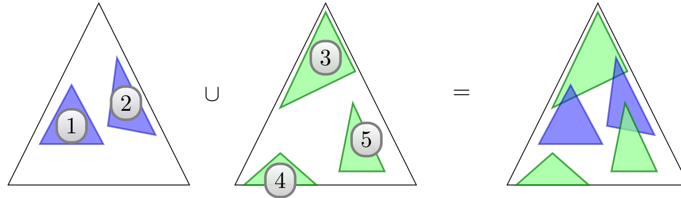


Figure 9: *Orthogonalization in a weakly orthogonal TRS.*

Consider for example Figure 9, where the redexes 2 and 3 overlap with each other. When trying to solve this overlap, we have to be careful since replacing the redex 2 by 3 as well as 3 by 2 creates new conflicts.

The case of finitary weakly orthogonal rewriting is treated in [9, Theorem 8.8.23]. There an inside-out algorithm is employed, consisting of inductively extending an orthogonalization of the subtrees to the whole tree. The basic observation is that one overcomes the difficulties pointed out above by starting at the bottom of the tree and solving overlaps by choosing the deeper (innermost) redex.

Example 6.2. We consider Figure 9 and apply the orthogonalization algorithm from [9, Theorem 8.8.23]. We start at the bottom of the tree. The first overlap we find is between the redexes 2 and 5; this is removed by replacing 2 with 5. Then the overlap between 2 and 3 has also disappeared. The only remaining overlap is between the redexes 3 and 1. Hence

we replace 3 by 1. As result we obtain two orthogonal multi-steps $\{1, 5\}$ and $\{1, 4, 5\}$.

Note that the above algorithm does not carry over to the case of infinitary multi-steps since we may have infinite chains of overlapping redexes and thus have no bottom to start at. This is illustrated on the right.

Example 6.3. As an example where such an infinite chain of overlaps arises we consider the TRS R consisting of the rule:

$$A(A(A(x))) \to A(x)$$

together with two multi-steps of blue and green redexes in the term A^ω:

$$A(A(A(A(A(A(A(A(A(A(A(A(A(A(A(\ldots)\ldots)$$

The blue redexes are marked by overlining, the green redexes by underlining.

Definition 6.4. A *cluster* is a non-empty set of redexes which forms a connected component with respect to $\leftrightsquigarrow$. The *pattern* of a cluster is the union of the patterns of its redexes. A cluster is a Y-*cluster* if it contains a pair of redexes at parallel positions (Figure 10, cases (ii) and (iv)); otherwise it is an I-*cluster* (Figure 10, cases (i) and (iii)). A Y-*redex* is a redex in a Y-cluster.

At first sight one might expect that Y-redexes are due to trivial rules of the form $\ell \to r$ with $\ell \equiv r$. However, the following example illustrates that, in general, this is not the case.

Example 6.5. Let R consist of the following (non-trivial) rules:

$$f(g(x, y), z, g(a, a)) \to f(g(y, x), z, g(a, a)) \qquad (\rho_1)$$

$$f(g(a, a), z, g(x, y)) \to f(g(a, a), z, g(y, x)) \qquad (\rho_2)$$

$$g(x, y) \to g(y, x) \qquad (\rho_3)$$

We consider the term $f(g(a, a), t, g(a, a))$ which contains both a ρ_1-redex and a ρ_2-redex at the root, a ρ_3-redex at disjoint positions 1 and 3. These redexes form a Y-cluster.

Notwithstanding the above example, it is always safe to drop Y-redexes from multi-steps without changing the outcome of the multi-step. This result is implicit in [5]. In particular in [5, Remark 4.38] it is mentioned that Y-clusters are a generalisation of Takahashi-configurations.

Lemma 6.6. *Let Y be a term in which the non-variable positions form the pattern of a* Y*-cluster, σ a substitution, and $Y\sigma \to s$ a step in Y. Then $s \equiv Y\sigma$ and subterms outside of Y have not been affected. (Note that subterms fully contained in the pattern of a* Y*-cluster* can *be affected.)*

Proof. By weak orthogonality redexes in a cluster have the same effect. Since Y-clusters have redexes at disjoint positions, it follows that contraction of any redex in a Y-cluster results in the same term. By applying this argument for the result of replacing the subterms outside of the Y-cluster by fresh variables, we conclude that none of these subterms can be affected (moved, copied, deleted) by contracting a redex from the Y-cluster. ∎

Lemma 6.7. *Let R be a weakly orthogonal TRS, $t \in Ter^\infty(\Sigma)$ a term. Let U be a set of non-overlapping redexes in t. Furthermore, let $V \subseteq U$ be such that every redex in V is contained in a* Y*-cluster of t. Then the multi-step with respect to $U \backslash V$ results in the same term as the multi-step with respect to U.*

Proof. We reduce in the complete development first all Y-redexes: by Lemma 6.6 this leaves the term as well as all redexes outside of Y-clusters untouched. As a consequence, the result of the complete development (multi-step) depends only on the redexes outside of Y-clusters. ∎

Definition 6.8. Let $\sigma : s \mathrel{-\!\!\circ\!\!\to}_U t_1$ and $\delta : s \mathrel{-\!\!\circ\!\!\to}_V t_2$ be multi-steps. An *orthogonalization witness* for the pair $\langle \sigma, \delta \rangle$ of multi-steps is a pair $\langle f_U, f_V \rangle$ of injective partial functions $f_U : U \rightharpoonup U \cup V$ and $f_V : V \rightharpoonup U \cup V$ such that it holds: (i) $ran(f_U)$ and $ran(f_V)$ are orthogonal sets of redexes in t; (ii) for all $u \in dom(f_U)$, $f_U(u) \leftrightsquigarrow u$, as well as, for all $v \in dom(f_V)$, $f_V(v) \leftrightsquigarrow v$; and (iii) $(U \setminus dom(f_U)) \cup (V \setminus dom(f_V)) \subseteq \{v : v \text{ is Y-redex in } t\}$.

Informally, an orthogonalization witness of multi-steps w.r.t. redex sets U and V defines (as stated in the proposition below) an orthogonalization consisting of multi-steps w.r.t. redex sets U' and V' that arise from U and V by exchanging redexes with equivalent, overlapping ones, and by possibly dropping some Y-redexes which have no effect.

Proposition 6.9. *Let $\sigma : s \mathrel{-\!\!\circ\!\!\to}_U t_1$ and $\delta : s \mathrel{-\!\!\circ\!\!\to}_V t_2$ be multi-steps, and let $\langle f_U, f_V \rangle$ be an orthogonalization witness for $\langle \sigma, \delta \rangle$. Then $U' = ran(f_U)$ and $V' = ran(f_V)$ are orthogonal sets of redexes in s, and there exist multi-steps $\sigma' : s \mathrel{-\!\!\circ\!\!\to}_{U'} t_1$ and $\delta' : s \mathrel{-\!\!\circ\!\!\to}_{V'} t_2$, and hence an orthogonalization $\langle \sigma', \delta' \rangle$ of $\langle \sigma, \delta \rangle$.*

We now define a top–down orthogonalization algorithm. Roughly speaking, we start at the top of the term and replace overlapping redexes with the outermost one. However, care has to be taken in situations as depicted in Figure 9.

Theorem 6.10. *Let R be a weakly orthogonal TRS, $t \in Ter^{\infty}(\Sigma)$ a (possibly infinite) term. Every pair $\langle \sigma, \delta \rangle$ of multi-steps $\sigma : t \mathrel{-\!\!\circ\!\!\to} t'$ and $\delta : t \mathrel{-\!\!\circ\!\!\to} t''$ has an orthogonalization.*

Proof. Let $\sigma : s \mathrel{-\!\!\circ\!\!\to}_U t_1$ and $\delta : s \mathrel{-\!\!\circ\!\!\to}_V t_2$ be multi-steps with respect to sets U and V of (non-overlapping) redexes. In view of Proposition 6.9 it suffices to construct an orthogonalization witness $\langle f_U, f_V \rangle$ for $\langle \sigma, \delta \rangle$. Briefly, we will show that it is always possible to solve outermost conflicts without creating fresh ones. After solving a conflict, the orthogonalization continues with the next conflict that is now at a top-most position.

If U and V are orthogonal, then we are finished (then f_U and f_V are both the identity). In this proof, by overlap we mean non-identical redexes whose patterns overlap. If there exist overlaps, let $u \in U \cup V$ be a topmost redex (that is, having minimal depth) among the redexes which have an overlap. Without loss of generality (by symmetry) we assume that $u \in U$ and let $v \in V$ be a topmost redex among the redexes in V overlapping u.

We distinguish the following cases:

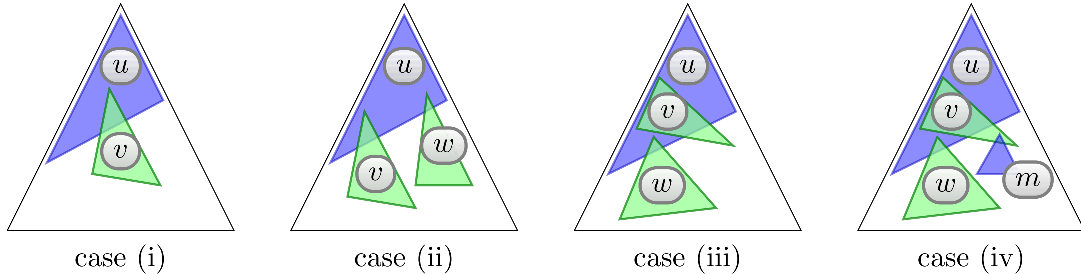


Figure 10: *Case distinction for the orthogonalization algorithm.*

(i) If v is the only redex in V that overlaps with u, case (i) of Figure 10, then we can safely

replace v by u. More precisely, we define $f_U(u) = u$ and $f_V(v) = u$ and continue the orthogonalization with $\langle U \setminus \{u\}, V \setminus \{v\}\rangle$, that is, the remaining redexes. Note that, since $(U \setminus \{u\}) \cup (V \setminus \{v\})$ contains no redexes overlapping u, the orthogonalization of the remainder cannot create overlaps with u.

Otherwise we pick a redex $w \in V$, $w \neq v$ and w overlaps u.

(ii) Assume that v and w are at disjoint positions, case (ii) of Figure 10. Then u, v and w are Y-redexes and can be dropped from U and V by Lemma 6.7. That is, we choose $f_U(u)$, $f_V(v)$ and $f_V(w)$ to be undefined, and continue the orthogonalization with the remainder $\langle U \setminus \{u\}, V \setminus \{v, w\}\rangle$.

Otherwise, v and w are not disjoint, and then w must be nested inside v.

(iii) If u is the only redex from U overlapping v, case (iii) of Figure 10, then we can replace u by v. That is, we define $f_U(u) = v$ and $f_V(v) = v$. We continue the orthogonalization with $\langle U \setminus \{u\}, V \setminus \{v\}\rangle$; that is *including* w since w may have further overlaps that need to be resolved.

(iv) In the remaining case there must be a redex $m \in U$, $m \neq u$ and m overlaps with the redex v, see case (iv) of Figure 10. We pick such an m. Since U and V are developments u cannot overlap with m, and v cannot overlap with w. We have that w is nested in v, both overlapping u, but m is below the pattern of u, overlapping v. Hence w and m must be at disjoint positions (v cannot tunnel through w to touch m); this has also been shown in [5]. Then u, m, v and w are contained in a Y-cluster, and hence they can be removed by Lemma 6.7. We choose $f_U(u)$, $f_U(m)$, $f_V(v)$, $f_V(w)$ to be undefined, and continue the orthogonalization with the remainder $\langle U \setminus \{u, m\}, V \setminus \{v, w\}\rangle$.

For all redexes $u \in U$ and $v \in V$ for which we have not specified $f_U(u)$ or $f_V(v)$, respectively, we define $f_U(u) = u$ or $f_V(v) = v$ (this concerns those u and v that either had no overlaps, or the conflicts have been solved by rearranging another redex positions). ∎

We obtain the diamond property as a corollary.

Corollary 6.11. *For every weakly orthogonal TRS without collapsing rules, (infinite) multi-steps have the diamond property.*

Proof. Let σ, δ be two coinitial complete developments $t_1 \stackrel{\sigma}{\leftarrow} s \stackrel{\delta}{\rightarrow} t_2$. Then by Theorem 6.10 there exists an orthogonalization $\langle \sigma', \delta' \rangle$ of σ, δ. The orthogonal projections σ'/δ' and δ'/σ' are complete developments (multi-steps) again, which are strongly convergent since the rules are not collapsing. Hence $t_1 \stackrel{\delta'/\sigma'}{\rightarrow} s' \stackrel{\sigma'/\delta'}{\leftarrow} t_2$. ∎

Note that in Corollary 6.11 the non-collapsingness is a necessary condition. To see this, reconsider Example 5.12 and observe that the non-confluent derivations are developments.

In a similar vein, we can prove the triangle property for infinitary weakly orthogonal multi-steps without collapsing rules:

Theorem 6.12. *For every weakly orthogonal TRS without collapsing rules, (infinite) multi-steps have the triangle property.* ∎

7. Conclusions

We have shown the failure of UN^∞ for weakly orthogonal TRSs in the presence of two collapsing rules. For weakly orthogonal TRSs without collapsing rules we established that

CR^∞ (and hence UN^∞) holds, and that this result is optimal in the sense that allowing only one collapsing rule is able to invalidate CR^∞.

However, the failure of UN^∞ for two collapsing rules raises the following question:

Question 7.1. Does UN^∞ hold for weakly orthogonal TRSs with *one* collapsing rule?

Furthermore, we have shown that infinitary developments in weakly orthogonal TRSs without collapsing rules have the diamond property. In general this property fails already in the presence of only one collapsing rule.

The following table summarizes the results of this paper (coloured green) next to known results (black).

		finitary				*infinitary*			
		PML	CR	UN	NF	PML^∞	CR^∞	UN^∞	NF^∞
first-order	OTRS	yes	yes	yes	yes	yes	no	yes	yes
	WOTRS	yes	yes	yes	yes	**yes**	no	**no**	**no**
	nc-WOTRS	yes	yes	yes	yes	**yes**	**yes**	**yes**	**yes**
	1c-WOTRS	yes	yes	yes	yes	**yes**	no	**?**	**?**
higher-order	$\lambda\beta$	yes	yes	yes	yes	no	no	yes	yes
	fe-OCRS	yes	yes	yes	yes	no	no	yes	yes[4]
	$\lambda\beta\eta$	yes[2]	yes	yes	yes	no	no	**no**	**no**
	WOCRS[1]	yes	yes	yes	yes	no	no	**no**	**no**

The nc-WOTRSs are weakly orthogonal TRSs without collapsing rules; 1c-WOTRSs likewise with one collapsing rule. The fe-OCRSs are fully extended orthogonal CRSs, see [4].

References

[1] H. Barendregt and J.W. Klop. Applications of Infinitary Lambda Calculus. *Inf. Comput.*, 207(5):559–582, 2009.

[2] I. Bethke, J.W. Klop, and de Vrijer, R.C. Descendants and Origins in Term Rewriting. *Inf. Comput.*, 159(1–2):59–124, 2000.

[3] R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Inf. Comput.*, 119(1):18–38, 1995.

[4] J. Ketema and J. Grue Simonsen. Infinitary Combinatory Reduction Systems: Confluence. *LMCS*, 5(4):1–29, 2009.

[5] J. Ketema, J.W. Klop, and V. van Oostrom. Vicious Circles in Rewriting Systems. CKI Preprint 52, Universiteit Utrecht, 2004. Available at `http://www.phil.uu.nl/preprints/aips/`.

[6] J.W. Klop and R.C. de Vrijer. Infinitary Normalization. In *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.

[7] P. Severi and F.-J. de Vries. An Extensional Böhm Model. In S. Tison, editor, *RTA 2002*, volume 2378 of *LNCS*, pages 159–173, 2002.

[8] P. Severi and F.-J. de Vries. Continuity and Discontinuity in Lambda Calculus. In *TLCA 2005*, volume 3461 of *LNCS*, pages 369–385, 2005.

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

[10] Vincent van Oostrom. Finite family developments. In Hubert Comon, editor, *Proceedings of RTA 1997*, volume 1232 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 1997.

[2]Beware: in [2] a counterexample is given to PML for $\lambda\beta\eta$, but that pertains to the stronger (classical) version of PML where the 'parallel move' has to consist of contractions of 'residuals' of the originally contracted redex.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 103-118
http://rewriting.loria.fr/rta/

THE UNDECIDABILITY OF TYPE RELATED PROBLEMS IN TYPE-FREE STYLE SYSTEM F

KEN-ETSU FUJITA [1] AND ALEKSY SCHUBERT [2]

[1] Gunma University, Tenjin-cho 1-5-1, Kiryu 376-8515, Japan
E-mail address: `fujita@cs.gunma-u.ac.jp`

[2] The University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland
E-mail address: `alx@mimuw.edu.pl`

Abstract. We consider here a number of variations on the System F, that are predicative second-order systems whose terms are intermediate between the Curry style and Church style. The terms here contain the information on where the universal quantifier elimination and introduction in the type inference process must take place, which is similar to Church forms. However, they omit the information on which types are involved in the rules, which is similar to Curry forms. In this paper we prove the undecidability of the type-checking, type inference and typability problems for the system. Moreover, the proof works for the predicative version of the system with finitely stratified polymorphic types. The result includes the bounds on the Leivant's level numbers for types used in the instances leading to the undecidability.

1. Introduction

The type systems can be viewed as formalisms in which reasonable properties of computational entities can be precisely formulated. The traditional approach distinguishes two kinds of computational entities considered in connection with type systems, i.e., Church-style λ-terms and Curry-style ones. The System F ($\lambda 2$) of Girard-Reynolds, however, allows of introducing intermediate families of terms between Church-style and Curry-style.

From the functional programming perspective, the Curry-style terms are interesting since they can serve as a useful notation to define programs with little notational overhead. However, more typing information can make the process of the program understanding easier. In order to broaden the scope of different compromise choices between these extremes, it is useful to study intermediate systems with different amounts of notational burden.

One family of such terms is the family of domain-free terms. It arose in [HS99] as a good target language for CPS transformations. This style corresponds to removing from the

1998 ACM Subject Classification: F.4.1.

Key words and phrases: Lambda calculus and related systems, type checking, typability, partial type inference, 2nd order unification, undecidability, Curry style type system, Church style type system, finitely stratified polymorphic types.

[2] This work was partly supported by the Polish government grant no N N206 355836.

© K. Fujita and A. Schubert
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.103

terms the information on which types were involved in the application of the λ abstraction rule. Type systems with quantification permit also removing the information on which types are involved in places where the quantification rules are used, but to leave the trace of the quantification rule itself. This gives rise to *type-free systems* which we consider here.

The type-checking, type inference and typability problems were thoroughly studied for the polymorphic lambda calculus and its various fragments and variations, e.g. [Boe85, Pfe93, Wel99, Sch98, FS00, KTU90b, Mai90]. These and other studies show that even a little bit more polymorphism than in ML gives rise to an undecidable system. It is, however, still unknown if the polymorphic lambda calculus in the predicative formulation (i.e. a system where types are divided into levels and a variable of a particular level can be substituted for by types of a lower level), the so called finitely stratified polymorphic lambda calculus [Lei91], is decidable with this regard.[1] The current paper gives a bound on the undecidable cases of the predicative system. The proof of the undecidability for the type-free $\lambda 2$, we present here, works in the system with limited level number (level 1 in case of type-checking and type inference, and level 2 in case of typability). This proof, however, cannot be adapted easily to the Curry-style $\lambda 2$ as it is based on a reduction from second-order unification, while the Curry-style requires a technique which is based on semiunification [Wel99, KTU90a].

Although this papers gives negative results, it provides an interesting perspective to the investigation of type systems as it presents a system with light-weight annotation burden, but which uses second-order unification as the basis for type reconstruction instead of the semiunification. This has the advantage that the research on higher-order unification is much more active and gives more opportunities for type systems with decidable type reconstruction procedures.

This paper is structured as follows. We introduce the type systems to deal with in Section 2. Section 3 is devoted to the undecidability of the restricted 2nd order unification problem. The undecidability proof for the type-free $\lambda 2$ systems is presented in Section 4.

2. Second-order Type Systems

We consider second-order polymorphic systems $\lambda 2$ of Girard-Reynolds in the type-free style.

2.1. Polymorphic System $\lambda 2$ (System F)

The polymorphic system $\lambda 2$ (System F) employs the connective $\rightarrow$ and second-order universal quantification $\forall$ to form expressions called $\lambda 2$-types:

$$A ::= X \mid (A \rightarrow A) \mid \forall X.A$$

The contexts of the system (written as $\Gamma, \Gamma', \Gamma_1$ etc.) are as usual finite mappings from term variables to types. The domain $\mathrm{Dom}(\Gamma)$ of a context Γ is the set of the term variables the context assigns types to. The terms of $\lambda 2$ in the *type-free style* are defined as follows. The

[1] To be precise, the construction of Wells [Wel99] gives rise to undecidable type-checking in the Leivant's level 2, the remaining cases i.e. type inference and typability, in general, and type-checking in level 1 are to our knowledge open.

system is an intermediate system between Church and Curry styles, where the symbols [] and Λ mark applications of the $\forall$ rules without the type information in terms.

$$M ::= x \mid (\lambda x.M) \mid (MM) \mid (\Lambda.M) \mid (M[])$$

The inference rules are as follows:

$$\frac{}{\Gamma, x\!:\!A \vdash_{\lambda 2\mathrm{tf}} x : A}\ (\mathrm{var})$$

$$\frac{\Gamma, x\!:\!A_1 \vdash_{\lambda 2\mathrm{tf}} M : A_2}{\Gamma \vdash_{\lambda 2\mathrm{tf}} \lambda x.M : A_1 \to A_2}(\to I) \quad \frac{\Gamma \vdash_{\lambda 2\mathrm{tf}} M_1 : A_1 \to A_2 \quad \Gamma \vdash_{\lambda 2\mathrm{tf}} M_2 : A_1}{\Gamma \vdash_{\lambda 2\mathrm{tf}} M_1 M_2 : A_2}(\to E)$$

$$\frac{\Gamma \vdash_{\lambda 2\mathrm{tf}} M : A}{\Gamma \vdash_{\lambda 2\mathrm{tf}} \Lambda.M : \forall X.A}\ (\forall I)^\star \qquad \frac{\Gamma \vdash_{\lambda 2\mathrm{tf}} M : \forall X.A}{\Gamma \vdash_{\lambda 2\mathrm{tf}} M[] : A[X := A_1]}\ (\forall E)$$

where $(\forall I)^\star$ denotes the eigenvariable condition $X \notin \mathrm{FV}(\Gamma)$.

The predicative version of the system divides the type variables into levels [Lei91].[2] We write $X^{(k)}$ to mark that X is in the level k. Then, the types in the level 0 and k for $k > 0$ are defined as follows:

$$\begin{array}{lcl} A^{(0)} & ::= & X^{(0)} \mid (A^{(0)} \to A^{(0)}) \\ A^{(k)} & ::= & X^{(k)} \mid A^{(k-1)} \mid (A^{(k)} \to A^{(k)}) \mid \forall X^{(k-1)}.A^{(k)} \end{array}$$

The minimum level of a type A is denoted as $\mathsf{lvl}(A)$. Now, the rules of the predicative type-free $\lambda 2$ are as usual except the rules for the quantification:

$$\frac{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} M : A}{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} \Lambda.M : \forall X^{(k)}.A}\ (\forall I)^\star \quad \frac{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} M : \forall X^{(k)}.A \quad \mathsf{lvl}(A_1) \le k}{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} M[] : A[X^{(k)} := A_1]}\ (\forall E)$$

where $(\forall I)^\star$ denotes the eigenvariable condition $X \notin \mathrm{FV}(\Gamma)$.

We say that a derivation of a judgement is done within level k, if for every judgement $\Gamma \vdash M : A$ in the derivation, we have $\mathsf{lvl}(A) \le k$ and $\mathsf{lvl}(B) \le k$ for all types B occurring in Γ.

On the other hand, $\lambda 2$ in the Curry style has well-known rules as follows:

$$\frac{\Gamma \vdash_{\lambda 2\mathrm{C}} M : A}{\Gamma \vdash_{\lambda 2\mathrm{C}} M : \forall X.A}\ (\forall I)^\star \qquad \frac{\Gamma \vdash_{\lambda 2\mathrm{C}} M : \forall X.A}{\Gamma \vdash_{\lambda 2\mathrm{C}} M : A[X := A_1]}\ (\forall E)$$

A predicative version of $\lambda 2$ in the Curry style is also defined for stratified types:

$$\frac{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{C}} M : A}{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{C}} M : \forall X^{(k)}.A}\ (\forall I)^\star \quad \frac{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{C}} M : \forall X^{(k)}.A \quad \mathsf{lvl}(A_1) \le k}{\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{C}} M : A[X^{(k)} := A_1]}\ (\forall E)$$

Then a type erasing map from terms in type-free style to those in Curry style is naturally defined so that $|\Lambda.M| = |M|$ and $|M[]| = |M|$. A type flatting map from stratified types to non-stratified ones is also defined so that $|X^{(k)}| = X$ and $|\forall X^{(k)}.A| = \forall |X^{(k)}|.|A|$. We have the following basic properties between Curry and type-free styles; and stratified and flatted types.

Proposition 2.1 (Erasing, lifting, flatting)**.** *Let a pair of systems* $(X, Y) = (\lambda 2\mathrm{tf}, \lambda 2\mathrm{C})$ *or* $(\mathrm{p}\lambda 2\mathrm{tf}, \mathrm{p}\lambda 2\mathrm{C})$. *Let* $(Z, W) = (\mathrm{p}\lambda 2\mathrm{tf}, \lambda 2\mathrm{tf})$ *or* $(\mathrm{p}\lambda 2\mathrm{C}, \lambda 2\mathrm{C})$.

[2]Levels should not be confused with type ranks [Lei83, KW94] defined as: $\mathrm{rank}(A) = 0$ when A contains no quantifier, $\mathrm{rank}(A_1 \to A_2) = \max(\mathrm{rank}(A_1) + 1, \mathrm{rank}(A_2))$, and $\mathrm{rank}(\forall X.A) = \mathrm{rank}(A)$. For instance, $(\forall X.(X \to X)) \to \forall Y.Y$ has rank 2, but has level 1 if $\mathsf{lvl}(X) = \mathsf{lvl}(Y) = 0$. Note also that the level here is different from the stratified level in [GR94].

Term\System	pλ2tf	λ2tf	pλ2C	λ2C
$(\lambda x.xx)(\lambda x.xx)$	No	No	No	No
$(\lambda x.xx)(\lambda x.xyx)$	No	No	No	Yes
$\lambda x.xx$	No	No	Yes	Yes
$\lambda x.x[]x$	No	Yes	-	-
$\lambda x.x[](x[])$	Yes	Yes	-	-

Figure 1: Typability for example terms in type-free and Curry style disciplines

(1) *If* $\Gamma \vdash_X M : A$ *then* $\Gamma \vdash_Y |M| : A$.
(2) *If* $\Gamma \vdash_Y M : A$ *then there exists an* X*-term* N *such that* $|N| \equiv M$ *and* $\Gamma \vdash_X N : A$.
(3) *If* $\Gamma \vdash_Z M : A$ *then* $\Gamma \vdash_W M : |A|$.

Notice that the systems λ2tf and pλ2tf, unlike λ2C and pλ2C, are syntax directed in the sense that the form of a term determines exactly which rule should be the last one in a derivation of its type. Still, in case of the $(\forall E)$, we do not know which types are used in the particular instance of the rule.

We show simple examples of terms in the systems on Figure 1, where "Yes" means typable but "No" untypable in the corresponding system (the second term is due to Urzyczyn [Lei91]).

The *type checking problem* (TCP) is the problem: given a term M, a type A, and a context Γ, is $\Gamma \vdash M : A$ derivable? The *type inference problem* (TIP) is the problem: given a term M and a context Γ, is there a type A such that $\Gamma \vdash M : A$ is derivable? Finally, the *typability problem* (TP) is the problem: given a term M, are there a context Γ and a type A such that $\Gamma \vdash M : A$ is derivable?

Note that in case of the predicative system we may consider each of the problems above in two versions: non-bounded ones (TCP, TIP, TP) in which we ask about the derivability in the predicative system in general and k-bounded ones (TCP^k, TIP^k, TP^k) in which we are given data that falls within the level k and want an answer if a derivation can be done within the level k exclusively.

In general, there is no direct relation between the undecidability of TCP (or TIP, or TP) and the undecidability of TCP^k. In particular the instances of TCP have no fixed level so the identical translation does not give the undecidability. The reduction in the other direction is not straightforward either as the systems do not enjoy the conservativity property.

Proposition 2.2. *For each* k *there are context* Γ*, term* M*, and a type* A *such that* Γ *and* A *are in level* k*,* $\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} M : A$ *is derivable, but there is no derivation for* $\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} M : A$ *in level* k*.*

Proof. We adapt the terms proposed by Urzyczyn (see [Lei91]). Let the term $M_1 \equiv \lambda x.x[](x[])$ and $M_{k+1} \equiv \lambda y.y[]M_k(y[])$. The term M_{k+1} is typable in level $k+1$ but is not typable in level k. Consider now the judgement $\vdash (\lambda x.\lambda y.y)M_{k+1} : X^{(k)} \to X^{(k)}$. This term is typable in level $k+1$ only, but is not typable in level k while the context and the type $X^{(k)}$ are in level k. ∎

However, a slightly weaker version of conservativity indeed holds for the systems. We say that a type-free $\lambda 2$ term M is in normal form if it has one of the following shapes:[3]

(1) $xM_1 \dots M_n$ $(n \geq 0)$ where M_i is either a term in normal form or [], or
(2) $\lambda x.M$ where M is in normal form, or
(3) $\Lambda.M$ where M is in normal form.

Proposition 2.3. *Let M be a term in normal form and Γ, A be in level k. If $\Gamma \vdash_{\mathrm{p}\lambda 2\mathrm{tf}} M : A$ is derivable then it is derivable in level k.*

Proof. Induction on the term M in normal form. ∎

2.2. Connections with Partial Type Reconstruction

The problem of partial type reconstruction is a crucial problem for functional programming languages. Consider the following situation. We would like to write a generic function which transforms an existing polymorphic function so that the resulting function prints out some textual information. Currently, a definition of such a function could look like as follows (in the syntax of OCaml)

```
# let addInfo = fun f x -> print "we call polymorphic"; f x;;
```

However, this is not possible in functional languages such as OCaml as the functional arguments cannot be polymorphic. Still, extension of the language to make such definitions possible is apparently useful. One of the drawbacks of the notation above is that it breaks the current convention that an application of a function is always monomorphic. Therefore, it is useful to warn a programmer that a particular function is polymorphic, for instance in the following fashion:

```
# let addInfo = fun f x -> print "we call polymorphic"; (f[]) x ;;
```

Since the functional programming languages give the programmer a freedom of giving the typing information wherever it is suitable for readability or comprehensiveness. Therefore, the functional programming languages require not only the procedures to decide the type inference, but stronger procedures that tackle the partial type reconstruction problem where some of the typing information is provided and some is omitted.

Along the lines of Boehm [Boe85], Pfenning [Pfe93] has proved that the partial type reconstruction problem for the pure polymorphic $\lambda 2$ is equivalent to the second-order unification problem as far as decidability is concerned. This means the problem is undecidable. He also proved the undecidability result for the predicative system. Our problem TIP for type-free $\lambda 2$ can be regarded as a restricted instance of the partial type reconstruction problem in which the instances cannot contain types in terms, but may contain placeholders where types should be instantiated in type derivations. The question of undecidability for terms in this form has been mentioned in the paper [Pfe93] and we confirm here that the problem is undecidable. Note that this strategy is very practical as the additional typing information in the terms is provided very rarely in real functional programs.

In [FS00], we demonstrated that TIP is undecidable for the predicative domain-free $\lambda 2$ by a reduction from the second-order unification problem. The problem TIP here for type-free $\lambda 2$ is a very restricted form of both problems. In general the proof methods applied in [Pfe93, FS00] do not work for the problem for the type-free style, since the previous

[3] The study of appropriate notion of reduction goes beyond the main topic of the paper.

methods essentially refer to type information which is to be erased in the type-free case. Even the direct application of the unification for flat forms to the construction of Pfenning does not bring the main result of the current paper.

It is worth pointing out that the results of [Pfe93, Boe85] indicate that it is impossible to devise an algorithm which allows the authors of functional programs to freely insert and omit the type annotations in the full polymorphic type discipline. We strengthen the result in such a way that already the strategy of showing where the second-order polymorphism is used, but omitting the information on how it is used gives rise to undecidability. Still, we hope that the existing decidable cases of the second-order unification can bring systems with decidable type related problems and the resulting systems will have the pragmatic advantage that the uses of polymorphisms are clearly marked in the source code of programs.

3. Undecidability of Restricted Unification

The proofs of undecidability for type-free system are done as a reduction of a strongly restricted version of the 2nd order unification problem to the problems. The version of the unification has been introduced in [FS09, FS]. The proof of the main theorem of the section is in [FS09].

We define here expressions for unification problems. We assume that a countably infinite set of type variables with level k is divided into three subsets: The first one contains first-order variables denoted by $X, Y, \ldots$, the second one constants denoted by $C, D, \ldots$, and the third one second-order variables denoted by $\mathsf{F}^n, \mathsf{G}^n, \ldots$ where the superscript n indicates their arity. The expressions we deal with in unification equations have the following form:

$$A, B \quad ::= \quad X \mid C \mid (A \to B) \mid \mathsf{F}^n A_1 \cdots A_n$$

Whenever it does not lead to confusion, we drop the superscript with arity. The expressions which do not contain second-order variables are called *first-order expressions*.

An instance of the unification problem consists of a set of equations

$$E = \{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}.$$

We say that the instance E is solvable if there exists a substitution S such that $S(A_1) = S(B_1), \ldots, S(A_n) = S(B_n)$. We write $\mathrm{Dom}(S)$ for the domain of S. A set of free variables in expressions of unification problems is defined as follows:

$$\mathrm{FV}_u(C) = \emptyset; \mathrm{FV}_u(X) = \{X\}; \mathrm{FV}_u(\mathsf{F}) = \{\mathsf{F}\}; \mathrm{FV}_u(A \to B) = \mathrm{FV}_u(A) \cup \mathrm{FV}_u(B).$$

We have to define a restricted second-order unification problem which can fit into the form of type constraints that arise in the type-free type system and is still undecidable. The basic idea here is to exploit the observation that the rules $(\forall E)$ work in a way similar to the application of a second-order expression to an argument. As the type-free systems omit from terms the information on an expression to which a second-order term is applied, we have to restrict the arguments of second-order variables so that the arguments can take up any form (monadic restriction below).

Definition 3.1 (Flat form)**.** An instance E of the unification problem is in *flat form* when it complies with all three restrictions below:

(1) *Root restriction*: Second-order variables occur only at root positions.

(2) *Monadic restriction*: If second-order variable occurs as $\mathsf{F}A_1 \cdots A_n$ then either all A_i are constants or all A_i are first-order variables. Moreover, the symbols A_i occur only in the equation where $\mathsf{F}A_1 \cdots A_n$ occurs.

(3) *Constant restriction*: Each time a second-order variable F is applied to a vector $X_1, \ldots, X_n$ of first-order variables as $\mathsf{F}X_1 \cdots X_n \doteq A$, there is a set of pairwise distinct constants $C_1, \ldots, C_n$ such that there is exactly one equation $\mathsf{F}C_1 \cdots C_n \doteq B \in E$, where $C_1, \ldots, C_n$ occur, all $C_1, \ldots, C_n$ occur in B, and the positions where the constants occur in B exist in A.

The idea similar to the one behind root restriction and monadic restriction can be found in [Ami90]. The constant restriction is necessary as it provides a clear indication on which positions correspond to the arguments of second-order variables. The presence of the constants can be simulated in the process of type inference by means of the $(\forall I)$ rules.

Theorem 3.2 (Undecidabiliy of unification with flat form)**.** *The problem of deciding if a given set of equations E in the flat form can be solved is undecidable.*

Proof. From a reduction of the unification of simple instances [Sch98, Sch01, LV00] to the unification of equations in the flat form. See [FS09, FS] for the details of the proof. ∎

The translation in the theorem above applied to the particular simple instances that give rise to the undecidability results in instances which contain 10 equations with second-order variables. Six of them involve one second-order variable F, while the other four another one G. To facilitate the understanding of the proofs below we present here the equations with second-order variables (omitting the other ones) that arise in the proof of Theorem 3.2:

$$\begin{array}{lcl}
\mathsf{F}X^1_1 \ldots X^1_n & \doteq & X_{\mathsf{F}A_1 \ldots A_n} \to A_1 \to \cdots \to A_n \to o \\
\mathsf{F}C^1_1 \ldots C^1_n & \doteq & X'_{\mathsf{F}A_1 \ldots A_n} \to C^1_1 \to \cdots \to C^1_n \to o \\[4pt]
\mathsf{F}X^2_1 \ldots X^2_n & \doteq & X_{\mathsf{F}A'_1 \ldots A'_n} \to A'_1 \to \cdots \to A'_n \to o \\
\mathsf{F}C^2_1 \ldots C^2_n & \doteq & X'_{\mathsf{F}A'_1 \ldots A'_n} \to C^2_1 \to \cdots \to C^2_n \to o \\[4pt]
\mathsf{F}X^3_1 \ldots X^3_n & \doteq & X_{\mathsf{F}A''_1 \ldots A''_n} \to A''_1 \to \cdots \to A''_n \to o \\
\mathsf{F}C^3_1 \ldots C^3_n & \doteq & X'_{\mathsf{F}A''_1 \ldots A''_n} \to C^3_1 \to \cdots \to C^3_n \to o \\[4pt]
\mathsf{G}X^4_1 \ldots X^4_m & \doteq & Y_{\mathsf{G}B_1 \ldots B_m} \to B_1 \to \cdots \to B_m \to o \\
\mathsf{G}C^4_1 \ldots C^4_m & \doteq & Y'_{\mathsf{G}B_1 \ldots B_m} \to C^4_1 \to \cdots \to C^4_m \to o \\[4pt]
\mathsf{G}X^5_1 \ldots X^5_m & \doteq & Y_{\mathsf{G}B'_1 \ldots B'_m} \to B'_1 \to \cdots \to B'_m \to o \\
\mathsf{G}C^5_1 \ldots C^5_m & \doteq & Y'_{\mathsf{G}B'_1 \ldots B'_m} \to C^5_1 \to \cdots \to C^5_m \to o
\end{array} \tag{3.1}$$

where F, G are fresh second-order variables of arity n, m, resp.; X with annotations and Y with annotations are fresh first-order variables; C^i_j with appropriate indices are subject to the constant restriction; and o is a distinguished type constant. We show here the equations in pairs (see case (3) in Definition 3.1).

4. Undecidability of Type Related Problems for Type-free System

Now, we embark on the reduction of the unification for the instances in the flat form to the type related problems. For a set of equations E we provide a λ-term M such that if a type derivation for M exists then a unifier S for E can be extracted from it. The main idea of the construction comes from [FS09, FS], and can be traced back to [Pfe93]. However, we

do not follow the latter construction in detail as we want to obtain a tight bound on type levels and avoid the occurrence of type variables in terms. The general idea of the approach here is that the shape of a type derived for a variable x_A occurring in M, related by the translation to a subexpression A in the set of equations E, strictly corresponds to the result of the substitution $S(A)$.

We simply write $A^n \to B$ for $A \to \cdots \to A \to B$ with n occurrences of A, and MN^n for $MN \cdots N$ with n applications to N.

Since we have a countably infinite set of term variables of λ-calculus, we can assume a one-to-one mapping between expressions of unification problems and term variables of λ-terms. Based on this, we write term variables x_A and y_A corresponding to an expression A of a unification problem. For instance, we have term variables x_X and y_X from a first-order variable X, and similarly term variables x_C and y_C from a constant C. In particular, the distinguished constant o gives rise to the term variable x_o.

4.1. Enforcing the Shape of First-order Terms

To achieve the goal sketched above, we have to provide a translation from instances of the second-order unification to terms which enforce particular forms of solutions. We start with enforcing of the shape of terms that do not involve second-order variables.

Terms which involve the variables associated with subexpressions of a given unification instance need to be put together in a single λ-term. To this end we use special variables that allow to glue terms. They are contained in a context

$$\Gamma_o = \{x_o : o, y_{o_1} : o \to o, \ldots, y_{o_n} : o^n \to o\}$$

for a fixed $n = 21$, where y_{o_i} are fresh term variables and o is the distinguished constant. We often shorten y_{o_i} to y_o when this does not lead to confusion. The following proposition holds.

Proposition 4.1. *There is a term M_o such that* $\mathrm{dom}(\Gamma_o) \subseteq \mathrm{FV}(M_o)$ *and for each Γ if $\Gamma \vdash M_o : A$ is derivable and Γ contains $x_o : o$ then $\Gamma_o \subseteq \Gamma$ and $A = o$.*

Moreover, if the level of o is $k \geq 0$ then the derivation can be done in level k.

Proof. Observe that a term $y_o x_o$ forces the argument of y_o to be of type o then $y_o(y_o x_o)$ forces also the result of y_o to be of type o so $y_o : o \to o$. The details of the construction and proof are left to the reader. ∎

The idea of enforcing employed in the sketch of the proof above can also be used below.

Definition 4.2 (Encoding of first-order expressions)**.** For a first-order expression A, we define a λ-term M_A, as follows:

(1) case $A \equiv X$ (first-order variable): the term $M_X \equiv y_o(y_X x_X)$,
(2) case $A \equiv C$ (constant): the term $M_C \equiv y_o(y_C x_C)$,
(3) case $A \equiv (A_1 \to A_2)$: the term
$$M_{A_1 \to A_2} \equiv y_o\,(y_{A_2}(x_{A_1 \to A_2} x_{A_1}))\,(y_{A_2} x_{A_2})\,M_{A_1}\,M_{A_2}.$$

Note that $\mathrm{FV}(M_A) \subseteq \mathrm{Dom}(\Gamma_o) \cup \{x_B, y_B \mid B \text{ is subexpression of } A\}$.

The encoding in Definition 4.2 is constructed in such a way that the types of variables x_A follow the structure of the corresponding expression A. In addition, the encoding gives enough freedom to enable the operation of first-order substitutions. This is precisely expressed by the following lemma.

Lemma 4.3. *For any substitution S, with $\Gamma \supseteq \Gamma_o$ defined by*

- $\Gamma(x_B) = S(B)$,
- $\Gamma(y_B) = S(B) \to o$

for all subexpressions B of A, we have that $\Gamma \vdash M_A : o$ is derivable.

Moreover, when variables in image of S are assigned level k then the derivation can be done in level k.

Proof. The proof is immediate by Definition 4.2 since $S(A_1 \to A_2) = S(A_1) \to S(A_2)$. ∎

We can now present the way how the first-order unification is encoded in λ-terms.

Definition 4.4 (Encoding of first-order unification)**.** Let E be a finite set of equations of first-order unification. In case $E = \emptyset$, we define $M_E \equiv x_o$. In case $E = \{A_1 \doteq B_1\} \cup E_0$, we define M_E to be:

$$M_E \equiv y_{o_5}\ (y_{A_1} x_{A_1})\ (y_{A_1} x_{B_1})\ M_{A_1}\ M_{B_1}\ M_{E_0}.$$

In the definition above, we use M_{A_1}, M_{B_1} to encode the shape of the expressions A_1, B_1 respectively. This is done in such a way that x_{A_1}, x_{B_1} have the types $S(A_1), S(B_1)$ for some substitution S. We can now force them to be equal by placing x_{A_1}, x_{B_1} as arguments to the same variable y_{A_1}. The rest of the set of equations can be taken into account in the same fashion in the subterm M_{E_0}. Note that

$$\mathrm{FV}(M_E) \subseteq \ \{x_B, y_B \mid B \text{ is subexpression of some expression in } E\} \cup \mathrm{Dom}(\Gamma_o).$$

The lemma below relates the solutions of first-order unification with derivations in the type-free System F.

Lemma 4.5. *Let E be a finite set of equations of first-order unification and S a substitution. The substitution S solves E if and only if there is a context $\Gamma \supseteq \Gamma_o$ such that $\Gamma \vdash M_E : o$ is derivable in level k and $\Gamma(x_B) = S(B)$, $\Gamma(y_B) = S(B) \to o$ for each subexpression B of expressions in E.*

Proof. The proof is by induction on the size of E. The case $E = \emptyset$ holds trivially. Consider now the case $E = \{A_1 \doteq B_1\} \cup E_0$.

($\Rightarrow$) Suppose that E is solvable under S, i.e., $S(A_1) = S(B_1)$ and S solves E_0. From Lemma 4.3, we have that $\Gamma_S \vdash M_{A_1} : o$ in level k and $\Gamma'_S \vdash M_{B_1} : o$ are derivable for some $\Gamma_S, \Gamma'_S \supseteq \Gamma_o$. By induction hypothesis we have in addition that $\Gamma''_S \vdash M_{E_0} : o$ is derivable. We may assume that domains of $\Gamma_S, \Gamma'_S, \Gamma''_S$ are minimal, i.e., they contain the domain of Γ_o and the free variables in M_{A_1}, M_{B_1}, M_{E_0} respectively. In this case, the intersection of domains of Γ_S, Γ'_S, Γ''_S contains only the domain of Γ_o and the variables of the form x_B, y_B. Therefore, each of the contexts assigns the same type to the variables. In this light we can let $\Gamma = \Gamma_S \cup \Gamma'_S \cup \Gamma''_S$ and preserve the derivability of $\Gamma \vdash M_{A_1} : o$, $\Gamma \vdash M_{B_1} : o$, and $\Gamma \vdash M_{E_0} : o$. Since S solves E we have $\Gamma(x_{A_1}) = S(A_1) = S(B_1) = \Gamma(x_{B_1})$, and therefore $\Gamma \vdash y_{A_1} x_{A_1} : o$ and $\Gamma \vdash y_{A_1} x_{B_1} : o$ are derivable. Now, we can easily assemble the derivation for $\Gamma \vdash M_E : o$.

($\Leftarrow$) Suppose that $\Gamma \vdash M_E : o$ is derivable (in level k) for some context $\Gamma \supseteq \Gamma_o$ with $\Gamma(x_B) = S(B)$, $\Gamma(y_B) = S(B) \to o$. This means that $\Gamma \vdash M_{E_0} : o$ is derivable. The induction hypothesis gives immediately that S solves E_0. We have also that $\Gamma \vdash M_{A_1} : o$ and $\Gamma \vdash M_{B_1} : o$ are derivable. As y_{A_1} is applied to both x_{A_1} and x_{B_1}, we have $\Gamma(x_{A_1}) = \Gamma(x_{B_1})$. Then we can infer that $S(A_1) = \Gamma(x_{A_1}) = \Gamma(x_{B_1}) = S(B_1)$. Hence all $A \doteq B \in E$ are solvable under S. ∎

4.2. Enforcing the Shape of Terms with Second-order Variables

Now that we know how to translate the equations with no second-order variables, we have to provide a translation for the equations that contain the variables. In order to make the presentation more concrete, we provide here a translation for the particular equations that result from the translation of the simple-instances to the equations in the flat form. Recall that the translation gives two pairs of equations for each second-order variable and the undecidable instances of the second-order unification of simple instances contain two variables F and G. We provide the following encoding for equations with G only, displayed in (3.1) on page 107. The case with F follows exactly the same pattern. See also Definition 3.1 and Theorem 3.2. We refrain from presentation of a general definition for equations with second-order variables as such a presentation is involved and obscures the main idea of the encoding. We believe that it is easier for a reader to understand the idea of the proof when it is presented in this more concrete fashion.

As a useful shorthand, we define a λ-term $M[]^{n+1} \equiv (M[])[]^n$ and $M[]^0 \equiv M$, which means successive application of $(\forall E)$. We also define a λ-term $\Lambda^{n+1}.M \equiv \Lambda^n.(\Lambda.M)$ and $\Lambda^0.M \equiv M$, which means successive application of $(\forall I)$.

Definition 4.6 (Encoding of second-order unification)**.** For a set of equations E such that the set $E\backslash E'$ consists of the equations in the flat form containing the second-order variable G of arity n:

$$\mathsf{G}X_1\cdots X_n \doteq B_1, \qquad \mathsf{G}C_1\cdots C_n \doteq B_2, \qquad \mathsf{G}Y_1\cdots Y_n \doteq B_3, \qquad \mathsf{G}C'_1\cdots C'_n \doteq B_4,$$

where $B_1 \equiv (X \to A_1 \to \cdots \to A_n \to o)$, $B_3 \equiv (Y \to A'_1 \to \cdots \to A'_n \to o)$,
$B_2 \equiv (X' \to C_1 \to \cdots \to C_n \to o)$, and $B_4 \equiv (Y' \to C'_1 \to \cdots \to C'_n \to o)$;
we define a λ-term M_E as

$$\begin{array}{ll} y_o & (y_{B_1}x_{B_1})\ (y_{B_1}(x_{\mathsf{G}}[]^n))\ (y_{B_2}x_{B_2})\ (y_{B_2}(x_{\mathsf{G}}[]^n)) \\ & (y_{B_3}x_{B_3})\ (y_{B_3}(x_{\mathsf{G}}[]^n))\ (y_{B_4}x_{B_4})\ (y_{B_4}(x_{\mathsf{G}}[]^n)) \\ & (y_{\mathsf{G}}x_{\mathsf{G}})\ (y_{\mathsf{G}}(\Lambda^n.\lambda z_1\ldots\lambda z_{n+1}.x_o))\ M_{B_1}\ M_{B_2}\ M_{B_3}\ M_{B_4}\ M_{E'} \end{array} \tag{4.1}$$

where x_o is as in Γ_o; $M_{B_1},\ldots,M_{B_4}$ are encodings of the expressions $B_1,\ldots,B_4$; and $M_{E'}$ is an encoding of the set E' of equations. In case $E = \emptyset$, $M_E = x_o$.

Note that

$$\begin{array}{rl} \mathrm{FV}(M_E) \subseteq & \{x_B, y_B \mid B \text{ is subexpression of some first-order expression in } E\}\cup \\ & \{x_{\mathsf{H}}, y_{\mathsf{H}} \mid \mathsf{H} \text{ is a second-order variable in } E\} \cup \mathrm{Dom}(\Gamma_o). \end{array}$$

Let us discuss a few issues concerning the construction of the term M_E. First note that $\Gamma_o(y_o) = o^{15} \to o$ and that all the 15 arguments from $(y_{B_1}x_{B_1})$ through $M_{E'}$ must have the same type o. Now, the variables y_D for $D \in \{B_1, B_2, B_3, B_4, \mathsf{G}\}$ are used to enforce that the left hand sides of the equations are equal to the right hand ones. The type of the variable x_{G} is supposed to be the result of the solution on the variable G. The monadic restriction allows us to simulate the work of variables $X_1,\ldots,X_n$ and $Y_1,\ldots,Y_n$ as well as the constants $C_1,\ldots,C_n$ and $C'_1,\ldots,C'_n$ by means of the type hole application $x_{\mathsf{G}}[]^n$. We exploit the knowledge of the shape of B_2, B_4 that we have from (3.1) on page 107 to enforce that the constants $C_1,\ldots,C_n$ and $C'_1,\ldots,C'_n$ are used in the quantifier eliminations when y_D is applied to $x_{\mathsf{G}}[]^n$ (where $D = B_1, B_2, B_3$ or B_4). The term to which y_{G} is applied enforces that the type of x_{G} has exactly n universal quantifiers in its head and $(n + 1)$

arrows inside. Note also that the term M'_E includes a similar encoding for the variable F. That encoding requires the use of $o^{21} \to o$ as we have 6 equations with F[4].

For a second-order substitution S we define $\Gamma_{o,S}$ as the smallest with regard to $\subseteq$ context such that $\Gamma_o \subseteq \Gamma_{o,S}$, $\Gamma_{o,S}(x_B) = S(B)$, and $\Gamma_{o,S}(y_B) = S(B) \to o$ for each subexpression B of a first-order expression in E, and $\Gamma_{o,S}(x_{\mathsf{G}}) = \forall X_1, \ldots, X_n.S(\mathsf{G})X_1 \cdots X_n$ and $\Gamma_{o,S}(y_{\mathsf{G}}) = (\forall X_1, \ldots, X_n.S(\mathsf{G})X_1 \cdots X_n) \to o$ for each second-order variable G occurring in E. Observe that it is possible to assign levels to type variables in $\Gamma_{o,S}$ so that the context as well as o are in level $k+1$ for each $k \geq 0$. We further work with such a version of the context.

Proposition 4.7. *Let E be the equations in the flat form above and S a substitution such that each type variable has level k. (1) If the substitution S solves E then there is a context $\Gamma \supseteq \Gamma_{o,S}$ such that $\Gamma \vdash M_E : o$ is derivable in level $k+1$. (2) If there is a context $\Gamma \supseteq \Gamma_{o,S}$ such that $\Gamma \vdash M_E : o$ is derivable in level $k+1$, then there is a substitution $S^\sharp$ such that $S^\sharp$ solves E and that $S^\sharp$ differs from S only on variables to which the second-order variables are applied.*

Proof. We assume here the notation as in Definition 4.6. The proof is by induction on the size of E. In case $E = \emptyset$ the claim holds trivially.

(1) Suppose that E is solvable under S, where each expression is constructed from type variables in level k. We show now that $\Gamma_{o,S} \vdash M_E : o$ is derivable in level $k+1$. First, observe that $\Gamma_{o,S}$ and o are in level $k+1$. Lemma 4.3 gives that M_{B_1}, M_{B_2}, M_{B_3}, and M_{B_4} are typable to o under appropriate contexts and as the contexts coincide with $\Gamma_{o,S}$ on the variables which occur free in the terms we may assume that the terms are typable to o under $\Gamma_{o,S}$. Similar reasoning starting with the use of Lemma 4.5 (in case E' is first-order) or the induction hypothesis (in case E' is second-order) gives that $\Gamma_{o,S} \vdash M_{E'} : o$ is derivable. Now, the derivability of $\Gamma_{o,S} \vdash y_D x_D : o$ for $D \in \{B_1, B_2, B_3, B_4, \mathsf{G}\}$ follows immediately from the definition of $\Gamma_{o,S}$. As the lemmas state, the derivations can be done in level $k+1$.

Now, for the proof of derivability of the terms in the form of $y_D(x_{\mathsf{G}}[]^n)$ for $D \in \{B_1, B_2, B_3, B_4\}$, we use in the type instantiations in places marked by [] that are indicated by appropriate equations, i.e. $S(X_1), \ldots, S(X_n)$; $C_1, \ldots, C_n$; $S(Y_1), \ldots, S(Y_n)$; $C'_1, \ldots, C'_n$, respectively. Then the derivability in level $k+1$ follows from the definition of $\Gamma_{o,S}$ and the fact that S unifies the equations displayed in Definition 4.6.

For the proof of derivability of $\Gamma_{o,S} \vdash y_{\mathsf{G}}(\Lambda^n.\lambda z_1 \ldots \lambda z_{n+1}.x_o) : o$, observe that $S(\mathsf{G}) = \lambda x_1 \ldots \lambda x_n.U_1 \to \cdots \to U_{n+1} \to B$ for some $U_1, \ldots, U_{n+1}$ and B. Otherwise it would be impossible to unify both the equation $\mathsf{G}X_1 \cdots X_n \doteq B_1$ and $\mathsf{G}C_1 \cdots C_n \doteq B_2$ by the constant restriction. Moreover, $B = o$ as the number of arguments is n and both B_1 and B_2 end with o. Therefore, we can assign the types $U_1, \ldots, U_{n+1}$ to $z_1, \ldots, z_{n+1}$ in $\lambda z_1 \ldots \lambda z_{n+1}.x_o$. Now, the derivability in level $k+1$ follows from the definition of $\Gamma_{o,S}$ and from the fact that all the steps above are done with help of types from $\Gamma_{o,S}$, the derivation is in level $k+1$.

Since all arguments of y_o are typable to o in $\Gamma_{o,S}$ and $y_o : o^{15} \to o$ is in Γ_o. We obtain a derivation for $\Gamma_{o,S} \vdash M_E : o$ in level $k+1$.

(2) Suppose, now, that $\Gamma \vdash M_E : o$ is derivable (in level $k+1$) for some $\Gamma \supseteq \Gamma_{o,S}$. As $\mathrm{FV}(M_E) = \mathrm{Dom}(\Gamma_{o,S})$, we can assume that $\Gamma_{o,S} \vdash M_E : o$. Now, we can prove that S is indeed a solution of E. Note first that either by induction hypothesis or by Lemma 4.5, S solves E'. As y_{G} is applied to both x_{G} and $\Lambda^n.\lambda z_1 \ldots \lambda z_{n+1}.x_o$, they have the same types. The type of $\Lambda^n.\lambda z_1 \ldots \lambda z_{n+1}.x_o$ (and at the same time the type of x_{G}) must be of

[4] $21 = 12 + 2 + 6 + 1$

the form $\forall X_1 \dots \forall X_n.(U_1 \to \dots \to U_{n+1} \to o)$ (*) (as the type system is syntax directed). From the derivability we obtain that $S(B_1) = S(\mathsf{G})D_1 \cdots D_n$ and $S(B_2) = S(\mathsf{G})D'_1 \cdots D'_n$ for some $D_1, \dots, D_n, D'_1, \dots, D'_n$, because of the way how y_{B_1}, y_{B_2} are used. Note that B_2 contains n positions (the positions of $C_1, \dots, C_n$) at which terms differ from the terms in the corresponding positions in B_1. Moreover, all these positions occur in the type marked as (*). Therefore, the only possible way to ensure that the equalities above hold is when variables occupy the positions in the type of x_{G}. Then it follows that the sequence $D'_1, \dots, D'_n$ is a permutation of $C_1, \dots, C_n$. We can now permute the application of the type instantiations so that the constants are applied in the order $C_1, \dots, C_n$. Then, the arguments $D_1, \dots, D_n$ must be permuted in the same way. Then, however, A_i must occur in $S(B_1) = S(\mathsf{G})D_{i_1} \cdots D_{i_n}$ where C_i is used in $S(B_2) = S(\mathsf{G})C_1 \cdots C_n$ for each fixed $i = 1, \dots, n$. This is because A_i occurs in B_1 in position where C_i occurs in B_2. In the end, we obtain that $S(B_1) = S(\mathsf{G})S(A_1) \cdots S(A_n)$ and $S(B_2) = S(\mathsf{G})C_1 \cdots C_n$. As variables $X_1, \dots, X_n$ occur only in $\mathsf{G}X_1 \cdots X_n$, we may change $S(X_l)$ to D_{i_l} and in this way obtain $S^{\sharp,1}$. A similar reasoning may be used to infer the equalities $S(B_3) = S(\mathsf{G})Y_1 \cdots Y_n$ and $S(B_4) = S(\mathsf{G})C'_1 \cdots C'_n$. In this way, we obtain the required solution $S^\sharp$ to the whole E. ∎

This ends the technical lemmas which are enough to prove undecidability of TCP and TIP.

4.3. Enforcing the Content of Contexts for TP

When TCP and TIP are considered, a context is a part of the initial data so we can provide one which is useful for the purpose of undecidability proof. This is no longer the case when the goal is the undecidability of TP. In particular, we have to make sure that the types of different constants used in unification expressions are indeed different. Moreover, the expressions from the substitution must not occur neither in the context nor in the resulting type. This can be obtained with the help of the following construction.

The first step is to turn the type variables which correspond to constants in the unification expressions into quantified variables. Let $\vec{x_C} = \{x_{C_1}, \dots, x_{C_m}\}$, where all the constants in E subject to the constant restriction are collected. Let $\vec{y} = (\mathrm{FV}(M_E) \cup \mathrm{FV}(M_o)) \setminus (\{x_o\} \cup \vec{x_C})$ and $\hat{M}_E$ be the following term:

$$\hat{M}_E \equiv f(\Lambda^{m+1}.\lambda x_{C_1}.\cdots \lambda x_{C_m}.\lambda x_o.\hat{K}(\lambda \vec{y}.y_o M_E M_o))$$

where M_o is as in Proposition 4.1 and $\hat{K} = \lambda x.g$. Note that $\mathrm{FV}(\hat{M}_E) = \{f, g\}$. Let

$$\Gamma_E = \{f : (\forall C_1^{(k)}.\cdots \forall C_m^{(k)}.\forall C_{m+1}^{(k)}.(C_{i_1} \to \cdots \to C_{i_{m+1}} \to W_1)) \to W_2,\ g : W_1\}$$

where W_1 and W_2 are any types in level $k+1$ that do not use $C_1, \dots, C_{m+1}$, $\Gamma_E(f)$ is in level $k+1$ and $i_1, \dots, i_{m+1}$ is a permutation of $1, \dots, m+1$.

With help of the term $\hat{M}_E$ we reduce the problem of making $C_1, \dots, C_m$ different to the problem to enforce the particular shape of a type for f. This is guaranteed by the following lemma.

Proposition 4.8. *Let E be the equations in the flat form above and $k \geq 0$. The problem E is solvable if and only if $\Gamma_E \vdash \hat{M}_E : W_2$ is derivable.*

Moreover, if E is solvable the derivation of $\Gamma_E \vdash \hat{M}_E : W_2$ can be done in level $k+1$.

Proof. ($\Rightarrow$) If E is solvable by S then we can use the substitution as in Proposition 4.7 to derive $\Gamma_{o,S} \vdash M_E : o$ in level $k+1$. As $\Gamma_o \subseteq \Gamma_{o,S}$, we can also derive $\Gamma_{o,S} \vdash M_o : o$ by Proposition 4.1. Since M_E, M_o are typable to o and $\Gamma_{o,S}$ contains types of level $k+1$ we can derive a type B for $M_* \equiv \lambda\vec{y}.y_o M_E M_o$ which is in level $k+1$. Let us define $\Gamma_C = \{x_C : C \mid x_C \in \vec{x_C}\} \cup \{x_o : o\}$. We can directly check that M_* is typable in $\Gamma_E \cup \Gamma_C$ and all of the derivation of $\Gamma_E \cup \Gamma_C \vdash M_* : B$ can be done in level $k+1$. Now, we can do the type inference for $\hat{K} = \lambda x.g : B \to W_1$ which allows us to derive $\Gamma_E \cup \Gamma_C \vdash \hat{K}M_* : W_1$. Then we can abstract all the variables from Γ_C and then do the type abstraction for the type variables that serve as constants i.e. $C_1, \dots, C_m, o$. Observe, that the way we use Proposition 4.7 means that $C_1, \dots, C_m, o$ are in level k. With this in mind we can see that it is possible to derive in level $k+1$ the type W_2 for $f(\Lambda^{m+1}.\lambda x_{C_1}.\cdots\lambda x_{C_m}.\lambda x_o.\hat{K}M_*)$, which is the required result.

($\Leftarrow$) Suppose that $\Gamma_E \vdash \hat{M}_E : W_2$ (in level $k+1$). From $\Gamma_E(f)$, we have

$$\Gamma_E \vdash \Lambda^{m+1}.\lambda x_{C_1}.\cdots\lambda x_{C_m}.\lambda x_o.\hat{K}(\lambda\vec{y}.y_o M_E M_o) : \forall C_1^{(k)}.\cdots\forall C_m^{(k)}.\forall C_{m+1}^{(k)}.(C_{i_1} \to \cdots \to C_{i_m} \to C_{i_{m+1}} \to W_1).$$

Then we have $\Gamma_E \vdash \lambda x_{C_1}.\cdots\lambda x_{C_m}.\lambda x_o.\hat{K}(\lambda\vec{y}.y_o M_E M_o) : C_{i_1} \to \cdots \to C_{i_m} \to C_{i_{m+1}} \to W_1$, where $C_1, \dots, C_{m+1}$ are fresh and pairwise distinct type variables from the variable condition of $(\forall I)$. Further, we obtain

$$\Gamma \equiv \Gamma_E \cup \{x_{C_1} : C_{i_1}, \dots, x_{C_m} : C_{i_m}, x_o : o\} \vdash \hat{K}(\lambda\vec{y}.y_o M_E M_o) : W_1,$$

and then we have $\Gamma \vdash \lambda\vec{y}.y_o M_E M_o : B'$ for some type B'. Hence, we can derive $\Gamma \vdash M_o : o$, which implies that $\Gamma_o \subseteq \Gamma$ by Proposition 4.1. We can now define a substitution S as $S(X) = B$ for all $x_X : B \in \Gamma$ and $S(\mathsf{H}) = \lambda x_1 \dots \lambda x_l.B[X_1 := x_1, \dots, X_l := x_l]$ for each second-order variable H in E such that $x_{\mathsf{H}} : \forall X_1 \dots \forall X_l.B \in \Gamma$. Observe now, that $\Gamma \supseteq \Gamma_{o,S}$ in the notation of Proposition 4.7 (up to inessential renaming of the constants). Then the case (2) of the proposition gives a slightly modified substitution $S^\sharp$ that solves E. ■

Now, we must enforce a particular form of type for the variable f. We define, therefore, a term N_f, which forces the required type. For this, let id $= \lambda x.x$ and ID $= \Lambda.\lambda x.x$. We write $\lambda^{n+1}x.M$ for $\lambda x.(\lambda^n x.M)$ where $\lambda^0 x.M \equiv M$. Let us define:

$$N_f \equiv z\,(z_1(fa))\,(z_2(a[]^{m+1}\,\mathrm{id}^{m+1}))\,(z_2(a[]^{m+1}\,\mathrm{ID}\,(\Lambda.\mathrm{ID})\dots(\Lambda^m.\mathrm{ID})))\,(z_3 a)\,(z_3(\Lambda^{m+1}.\lambda^{m+1}v.z_4))$$

where $z, z_1, a, z_2, z_3, v, z_4$ are fresh term variables.

Lemma 4.9. (a) *If N_f is typable where the derivation contains types only in level $(k+2)$ with $k \geq 0$, then for all types $D_1, \dots, D_6$ in level $(k+2)$ there is a context Γ_f such that*

$$\begin{array}{l} \Gamma_f(a) = \forall X_1^{(l_1)}.\cdots\forall X_{m+1}^{(l_{m+1})}.(X_1 \to \cdots \to X_m \to D_1) \text{ with} \\ \qquad X_1, \dots, X_{m+1} \notin \mathrm{FV}(D_1), \quad l_1, \dots, l_{m+1} \leq k+1, \\ \Gamma_f(f) = \Gamma_f(a) \to D_2, \qquad \Gamma_f(z_1) = D_2 \to D_3, \qquad \Gamma_f(z_2) = D_1 \to D_4, \\ \Gamma_f(z_3) = \Gamma_f(a) \to D_5, \Gamma_f(z_4) = D_1, \text{ and} \\ \Gamma_f(z) = D_3 \to D_4 \to D_4 \to D_5 \to D_5 \to D_6. \end{array} \tag{4.2}$$

(b) *Suppose that $D_1, \dots, D_6$ are types in level $k+2$. A judgement $\Gamma_f \vdash N_f : D_6$ is derivable in level $k+2$, where Γ_f is defined as in (4.2).*

Proof. (a) If N_f is typable in level $(k+2)$ with $k \geq 0$, then the subterm $(z_1(fa))$ enforces an arrow type on type of f. Further the pair of subterms $(z_3 a)$ and $(z_3(\Lambda^{m+1}.\lambda^{m+1}v.z_4))$ enforce the shape $\forall X_1^{(l_1)}.\cdots\forall X_{m+1}^{(l_{m+1})}.(A_1 \to \cdots \to A_m \to D_1)$, with $l_1,\ldots,l_{m+1} \leq k+1$ and some types A_i, D_1, on the type of a (being the argument type of f). Moreover, the subterms $(a[]^{m+1}\,\mathrm{id}^{m+1})$ and $(a[]^{m+1}\,\mathrm{ID}\,(\Lambda.\mathrm{ID})\ldots(\Lambda^m.\mathrm{ID}))$ enforce the shape

$$\forall X_1^{(l_1)}.\cdots\forall X_{m+1}^{(l_{m+1})}.(X_{i_1} \to \cdots \to X_{i_{m+1}} \to D_1)$$

on type of a for some permutation $i_1,\ldots,i_{m+1}$ of $1,\ldots,m+1$. In addition, the typability of $(\Lambda^{m+1}.\lambda^{m+1}v.z_4)$ gives $X_1,\ldots,X_{m+1} \notin \mathrm{FV}(D_1)$ from the variable condition on $(\forall I)$, so that we have

$$\Gamma_f(f) = (\forall X_1^{(l_1)}.\cdots\forall X_{m+1}^{(l_{m+1})}.(X_{i_1} \to \cdots \to X_{i_{m+1}} \to D_1)) \to D_2.$$

For types of other free variables, the analysis is straightforward.

(b) A direct check gives us the case of the lemma. ∎

4.4. The Main Theorem

We can now prove the main result of the paper.

Theorem 4.10 (TP, TIP, TCP)**.** TP^2, TIP^1*, and* TCP^1 *together with* TP*,* TIP*, and* TCP *are undecidable for the type-free system of* $\lambda 2$.

Proof. Consider first TIP^1. We can take as an instance of the problem $M \equiv \lambda\vec{y}.y_o M_E M_o$ and $\Gamma \equiv \Gamma_o \cup \{x_C : C \mid C \text{ is a constant}\}$ where all the type variables in Γ are in level 1 and $\vec{y}$ are all free term variables in M_E except those in Γ. If $\Gamma \vdash M : A$ is derivable in level 1 then there is Γ' such that $\Gamma' \vdash M_E : A'$ is derivable in level 1. By Proposition 4.1 applied to M_o we obtain $A' = o$. Now, we can define a substitution S such that $S(X) = B$ for all $x_X : B \in \Gamma'$ and $S(\mathsf{H}) = \lambda x_1 \ldots x_l.B[X_1 := x_1,\ldots,X_l := x_l]$ for each second-order variable H in E such that $x_{\mathsf{H}} : \forall X_1 \ldots \forall X_l.B \in \Gamma'$. Immediate check verifies that, $\Gamma' \supseteq \Gamma_{o,S}$ (in notation of Proposition 4.7). Now, Proposition 4.7(2) gives that E is solvable.

If E is solved by S then we can impose that nullary symbols are in level 0. Then Proposition 4.7(1) gives derivation of $\Gamma_{o,S} \vdash M_E : o$ in level 1 and Proposition 4.1 gives $\Gamma_{o,S} \vdash M_o : o$. These two can be immediately combined into a derivation of $\Gamma \vdash M : A$ for some A. In this way we reduced the solvability of equations in flat form to TIP^1. The same proof works for the unbounded version of TIP and for the impredicative system.

The reasoning for TCP^1 follows the same lines, but we take as an instance the judgement $\Gamma \vdash (\lambda v.x_o)(\lambda\vec{y}.y_o M_E M_o) : o$, where v is fresh and Γ is the context given for the TIP^1 above.

As for the proof that $\mathrm{TP}^{(2)}$ is undecidable, we can assume w.l.o.g that $\mathrm{FV}(N_f) \cap \mathrm{FV}(\hat{M}_E) = \{f\}$. We show now that the typability of $zN_f\hat{M}_E$ in level 2 is equivalent to solvability of E. Indeed, suppose that the judgement $\Gamma \vdash zN_f\hat{M}_E : A$ is derivable in level 2 for some Γ, A. From Lemma 4.9(a),

$$\Gamma(f) = (\forall X_1^{(1)}.\cdots\forall X_{m+1}^{(1)}.(X_1 \to \cdots \to X_m \to D_1)) \to D_2$$

with $X_1,\ldots,X_{m+1} \notin \mathrm{FV}(D_1)$. Hence, the shape of $\hat{M}_E$ implies that $\Gamma \vdash \hat{M}_E : D_2$ is derivable. By Proposition 4.8, we obtain the solvability of E.

In case E is solvable, we combine Lemma 4.9(b) with Proposition 4.8 in a straightforward way. Notice that $\hat{M}_E$ is typed here in level 2.

We can now conclude that TP is undecidable in level 2. The same proof works for the unbounded version of TP and for the impredicative system. ∎

A careful reader might spot that the solution S obtained from a derivation can contain occurrences of $\forall$ which is not used in standard second-order unification expressions. This can, however, be mitigated by the following proposition.

Proposition 4.11. *If a set E of equations does not contain occurrence of the symbol $\forall$ then for each solution S of E there is a solution which does not use $\forall$.*

Proof. In order to prove the proposition, you should simply fix a constant (or a variable) C and each time $S(X)$ or $S(\mathsf{H})$ contains an occurrence of $\forall$ replace the subterm at that occurrence with C. In this way we obtain a substitution S'. For each equation $A \doteq B$ we obtain that $S'(A) = S'(B)$ as each time we have something different in $S'(A)$ than in $S(A)$ this must be C. This means that $S(B)$ at the same position has $\forall$. Since $\forall$ does not occur in B it must occur in a term which comes from S. Then this occurrence of $\forall$ is replaced with C in S'. ∎

5. Concluding Remarks

The current paper shows new paths of investigation on the type-free systems, interesting type systems for functional programming with moderate type annotation and the relation to the second-order unification. We have proved the undecidability of the type-checking, type inference and typability problems for the predicative version of the System F in the type-free style. The proof method even works for the impredicative version by flatting stratified types. Namely, TCP, TIP, and TP are all undecidable for the System F in the type-free style. Then, as in [NTKN08], the technique of CPS-translation can be applied to show that TCP, TIP, and TP are all undecidable for the existential system $\lambda^{\exists}$ [FS09] in the type-free style, consisting of $(\neg, \wedge, \exists)$. In [FS09], we proved that all of the type related problems are in general undecidable for the type-free system of $\lambda^{\exists}$ consisting of $(\rightarrow, \exists)$. We remark that a detailed analysis on the proof method in [FS09] reveals that TCP and TIP are still undecidable for the finitely stratified $\lambda^{\exists}$ of $(\rightarrow, \exists)$ in level 2 and that TP is undecidable for the system in level 3 as well. Moreover, the extended version [FS] leads to stricter borders, such that TCP and TIP are undecidable in level 1 and TP is undecidable in level 2 for the predicative system $\lambda^{\exists}$ of $(\rightarrow, \exists)$.

For the predicative version of the System F in the type-free style, our results provide a strict undecidability border for TCP and TIP problems as they are undecidable for level 1 types (level 0 types have no quantifiers so they are equivalent to the simply typed lambda calculus). We also prove undecidability for TP in level 2. We believe that the current construction can be adapted to prove undecidability of TP in level 1. In that case, however, the constants obtained in the proof of Proposition 4.8 must be simulated by more complicated (arrow) types which makes the whole construction much more involved.

References

[Ami90] Gilles Amiot. The undecidability of the second order predicate unification problem. *Archive for Mathematical Logic*, 30(3):193–199, May 1990.

[Boe85] H.-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.

[FS] K. Fujita and A. Schubert. Existential type systems between Church and Curry style. Submitted. Available from `http://www.mimuw.edu.pl/~alx/papers/existential-chcu.pdf`.

[FS00] K. Fujita and A. Schubert. Partially typed terms between Church-style and Curry-style. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, number 1872 in LNCS, pages 505–520, 2000.

[FS09] K. Fujita and A. Schubert. Existential type systems with no types in terms. In P.-L. Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009, Proceedings*, number 5608 in LNCS, pages 112–126, 2009.

[GR94] Paola Giannini and Simona Ronchi Della Rocca. A type inference algorithm for a stratified polymorphic type discipline. *Information Computation*, 109(1–2):115–173, 1994.

[HS99] John Hatcliff and Morten Heine B. Srensen. CPS translations and applications: The cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, September 1999.

[KTU90a] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476, New York, NY, USA, 1990. ACM.

[KTU90b] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXTIME-complete. In *CAAP '90: Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, number 431 in LNCS, pages 206–220, London, UK, 1990. Springer-Verlag.

[KW94] Assaf J. Kfoury and Joeseph B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, New York, NY, USA, 1994. ACM.

[Lei83] Daniel Leivant. Polymorphic type inference. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, New York, NY, USA, 1983. ACM.

[Lei91] D. Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, 1991.

[LV00] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Information and Computation*, 159(1–2):125–150, 2000.

[Mai90] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 382–401, New York, NY, USA, 1990. ACM.

[NTKN08] K. Nakazawa, M. Tatsuta, Y. Kameyama, and H. Nakano. Undecidability of type-checking in domain-free typed lambda-calculi with existence. In *CSL '08: Proceedings of the 22nd International Workshop on Computer Science Logic*, number 5213 in LNCS, pages 478–492, Berlin, Heidelberg, 2008. Springer-Verlag.

[Pfe93] F. Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, September-October 1993.

[Sch98] Aleksy Schubert. Second-order unification and type inference for Church-style polymorphism. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 279–288, New York, NY, USA, 1998. ACM.

[Sch01] Aleksy Schubert. *Zastosowanie unifikacji do problemw wyprowadzania typw.* PhD thesis, The University of Warsaw, 2001. English title: Application of the unification to type inference problems, Polish text available from `http://www.mimuw.edu.pl/~alx/ftp-public/doktorat.pdf`.

[Wel99] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 119-134
http://rewriting.loria.fr/rta/

ON (UN)SOUNDNESS OF UNRAVELINGS

KARL GMEINER [1] AND BERNHARD GRAMLICH [1] AND FELIX SCHERNHAMMER [1]

[1] Faculty of Informatics, TU Wien, Austria
E-mail address: {gmeiner,gramlich,felixs}@logic.at

Abstract. We revisit (un)soundness of transformations of conditional into unconditional rewrite systems. The focus here is on so-called unravelings, the most simple and natural kind of such transformations, for the class of normal conditional systems without extra variables. By a systematic and thorough study of existing counterexamples and of the potential sources of unsoundness we obtain several new positive and negative results. In particular, we prove the following new results: Confluence, non-erasingness and weak left-linearity (of a given conditional system) each guarantee soundness of the unraveled version w.r.t. the original one. The latter result substantially extends the only known sufficient criterion for soundness, namely left-linearity. Furthermore, by means of counterexamples we refute various other tempting conjectures about sufficient conditions for soundness.

1. Introduction

1.1. Background and Motivation

Conditional term rewriting systems (CTRSs) are a very natural, though non-trivial and complex extension of unconditional ones (TRSs). This concerns both the theoretical foundations as well as applications and implementations of such systems. A well-studied approach to dealing with conditional rewriting is via transformation to unconditional systems such that the resulting unconditional system can simulate the original conditional one in an appropriate manner. Various transformations have been developed for that purpose. It is well-known that completeness of these transformations is easy to achieve and usually holds, whereas soundness is much harder to obtain and typically does not hold without imposing further conditions, e.g., restrictions on the rewrite relation in the resulting unconditional system. Informally, by *(simulation) soundness* we mean that whenever an original term reduces to another original term in the transformed system, then such a reduction is also possible in the original system. *(Simulation) completeness* is the dual property.

The above unsoundness phenomenon was discovered by Marchiori ([9, 8]) for the case of so-called unravelings,[1] but is also present in virtually all other known transformation

The first author was supported by a grant of the *Vienna PhD School of Informatics*, the last author by a grant of the *Austrian Academy of Sciences* (ÖAW-DOC grant No. 22361).

[1] The very idea of *unravelings* is actually much older and appears already e.g. in [4], though in a specialized form (for function definitions).

© K. Gmeiner, B. Gramlich, and F. Schernhammer
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.119

approaches. Approaches to more faithfully simulating rewriting in a conditional system via restricted rewriting in a transformed unconditional system include: *conditional eagerness* ([19], [16]), *innermost rewriting* ([15]), *membership conditional* and *context-sensitive* rewriting ([18], [14, 11, 13], [5], [17])). Yet, in all these approaches the imposed restrictions on rewriting in the transformed unconditional rewrite relation are a major source of complications for reasoning over and deriving properties of the respective transformation approaches. Hence, a deeper knowledge about the borderline between unsoundness and soundness would help to identify cases (classes of initial conditional systems) where soundness is guaranteed even for unrestricted rewriting in the transformed unconditional system. In such cases, one can safely use (unrestricted) rewriting in the transformed system, thus facilitating the analysis and implementation of the respective transformation. These are the main goals of the analysis that we are going to present in this paper.

1.2. Overview and Outline

We focus on the most basic class of conditional systems without extra variables, normal 1-CTRSs. This is motivated by the fact that even for these systems the analysis is rather non-trivial and properly understanding this case appears to be indispensable for later extending the results to other and more general classes of CTRSs. Furthermore, the focus is also restricted to unravelings, the most simple and intuitive class of transformations from CTRSs into TRSs. Again, simplicity and the goal of properly understanding the essential source(s) of unsoundness is the main motivation for this restriction. We expect that a substantial part of the analysis can also be reused for other transformation approaches for CTRSs.

The main contributions of the paper are as follows. Starting from an analysis of existing counterexamples to the unsoundness of unravelings we prove that each of the following conditions on a given normal 1-CTRS is sufficient for soundness of its unraveled version:

- confluence (Theorem 3.12)
- non-erasingness (Theorem 3.16)
- groundness of all conditions (Theorem 3.17)
- *weak left-linearity* (Theorem 3.33).

Especially interesting and practically relevant are the first criterion and the last one which substantially extends the only known criterion for soundness, left-linearity (cf. [8, 9]). In essence, *weak left-linearity* (cf. Definition 3.22) weakens left-linearity by allowing unconditional non-left-linear rules provided that variables that appear non-linear in the left-hand side do not appear at all in the right-hand side.

On the negative side, we disprove various other tempting conjectures about the sufficiency of conditions for soundness, regarding e.g. non-overlappingness, non-collapsingness and right-linearity.

The rest of the paper is structured as follows. After the preliminaries in Section 2, where we introduce unravelings and basic projection functions used later on, we develop the analysis in the main Section 3. Before concluding, the results obtained, potential extensions, open problems and related work are finally discussed in Section 4. Due to lack of space, some proofs are only sketched or omitted.[2]

[2]Missing and completed proofs can be found in the full technical report version of this paper, cf. `http://www.logic.at/staff/{gmeiner,gramlich,schernhammer}/`.

2. Preliminaries

We assume familiarity with the basic concepts and notations of abstract reductions systems (ARSs) and (conditional) term rewriting systems (CTRSs) (cf. e.g. [1], [3]). For the sake of readability we recall some notions and notations here. Moreover, we use the typical abbreviations for properties of rewrite systems, such as CR, NF, UN, $UN^{\rightarrow}$, ...

The set of (non-variable, variable) positions of a term s is denoted as $Pos(s)$ ($FPos(s)$, $VPos(s)$). By $root(s)$ we denote the root symbol of the term s. Throughout the paper $\mathcal{V}$ denotes a countably infinite set of variables and x, y, z denote variables from $\mathcal{V}$. By $Var(s)$ we denote the set of variables of a term s. Moreover $\overrightarrow{Var(s)}$ denotes the sequence of variables obtained by arranging the variables of $Var(s)$ in an arbitrary but fixed order.

A term rewriting system $\mathcal{R}$ is a pair $(\mathcal{F}, R)$ of a signature and a set of rewrite rules over this signature. Slightly abusing notation we also write $\mathcal{R}$ instead of R (leaving the signature implicit).

We denote a rewrite step from a term s to a term t at position p with respect to a rewrite system $\mathcal{R}$ and with a rule δ from $\mathcal{R}$ as $s \rightarrow_{p,\mathcal{R},\delta} t$. We also write $s \rightarrow t$ ($s \rightarrow_p t$ resp. $s \rightarrow_{p,\mathcal{R}} t$) if the position, rewrite system and applied rule (the rewrite system and applied rule resp. the applied rule) are clear from the context or of no relevance. Parallel reduction is denoted by $\mathrel{+\!\!\!\!\rightarrow}$ and $\rightarrow^{\leq 1}$ means reduction with one or zero steps.

The set of *one-step descendants* of a (subterm) position p of a term t w.r.t. a (one-step) reduction $t = C[s]_p \rightarrow_q t'$ is the set of subterm positions in t' given by

- $\{p\}$, if $q \geq p$ or $q \mid\mid p$,
- $\{q.o'.p' \mid t|_q = l\sigma, l|_o \in Var(l), q.o.p' = p, l|_o = r|_{o'}\}$, if $q < p$ and (a superterm of) s is bound to a variable in the matching of $t|_q$ with the left-hand side of the applied rule, and
- $\emptyset$, otherwise.

Slightly abusing terminology, when $t = C[s]_p \rightarrow_q t'$ with set $\{p_1, \ldots, p_k\}$ of one-step descendants in t', we also say that $t|_p$ has the one-step descendants $t'|_{p_i}$ in t'. The *descendant relation* (w.r.t. given derivations) is obtained as the (reflexive-)transitive closure of the one-step descendant relation. The relation of (one-step) *ancestors* of a subterm position (w.r.t. a given reduction sequence) is the inverse relation of the (one-step) descendant relation.

A conditional term rewriting system $\mathcal{R}$ (over some signature $\mathcal{F}$) consists of rules $l \rightarrow r \Leftarrow c$ where c is a conjunction of equations $s_i = t_i$. Equality in the conditions may be interpreted (recursively) e.g. as $\leftrightarrow^*$ (semi-equational case), as $\downarrow$ (join case), or as $\rightarrow^*$ (oriented case). In the latter case, if all right-hand sides of conditions are ground terms that are irreducible w.r.t. the unconditional version $\mathcal{R}_u = \{l \rightarrow r \mid l \rightarrow r \Leftarrow c \in \mathcal{R}\}$ of R, the system is said to be a *normal* one.

According to the distribution of variables, a conditional rule $l \rightarrow r \Leftarrow c$ may satisfy (1) $Var(r) \cup Var(c) \subseteq Var(l)$, (2) $Var(r) \subseteq Var(l)$, (3) $Var(r) \subseteq Var(l) \cup Var(c)$, or (4) no variable constraints. If all rules of a CTRS $\mathcal{R}$ are of type (i), $1 \leq i \leq 4$, we say that $\mathcal{R}$ is an i-CTRS. Given a conditional rewrite rule $l \rightarrow r \Leftarrow c$ and a variable x such that $x \in Var(r)$ but $x \notin Var(l)$, we say that x is an *extra variable.*

There exists abundant literature on transforming CTRSs into unconditional systems such that the original system can be appropriately simulated via reduction in the unconditional transformed one. For a unified parametrized approach to such transformations and the relevant terminology we refer to [6]. Unravelings as introduced and investigated in [8, 9] are the most simple and intuitive ones.

Definition 2.1 ((simultaneous) unraveling for normal 1-CTRSs ([9, 8], cf. also [15]))**.** Given a normal 1-CTRS $\mathcal{R} = (\mathcal{F}, R)$, every conditional rule

$$\delta\colon l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n$$

of $\mathcal{R}$ is transformed into[3]

$$\begin{array}{rcll} l & \to & U^\delta(s_1, \ldots, s_n, \overrightarrow{Var(l)}) & \quad (\textit{introduction rule}) \\ U^\delta(t_1, \ldots, t_n, \overrightarrow{Var(l)}) & \to & r & \quad (\textit{elimination rule}) \end{array}$$

Unconditional rules remain invariant. The resulting (*unraveled*) TRS is denoted as $U(\mathcal{R})$ or $\mathcal{R}'$ (over the signature $\mathcal{F}' = \mathcal{F} \cup \{U^\delta \mid \delta\colon l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n \in \mathcal{R}\}$). Instead of the new symbols U^δ (corresponding to rule δ) we sometimes use other ones if appropriate.

Symbols from $\mathcal{F}' \setminus \mathcal{F}$ are also called U-symbols. Terms rooted by such symbols are called U-terms or U-rooted terms. Every U-symbol corresponds to a particular conditional rewrite rule of the original CTRS according to Definition 2.1. Hence, we write U^δ to indicate that U^δ corresponds to the rewrite rule δ. Moreover, if there is only one conditional rule defining a function symbol f we may also write U^f to identify this rule. Henceforth, $\mathcal{R}$ denotes a normal 1-CTRS unless stated otherwise.

The signature of an unraveled CTRS $\mathcal{R}'$ is a superset of the signature of the CTRS $\mathcal{R}$. Hence, terms in $\mathcal{R}'$-reductions are terms over this extended signature in general (we also call them *mixed* terms). Throughout the paper, when dealing with CTRSs $\mathcal{R} = (\mathcal{F}, R)$ we denote by $\mathcal{R}'$ the corresponding unraveled TRS, by $\mathcal{F}'$ the extended signature of the TRS, by $\mathcal{T}$ the terms over the signature $\mathcal{F}$ and by $\mathcal{T}'$ the terms over the extended signature $\mathcal{F}'$. For proof-theoretical reasons, in particular to show that unraveled systems are not too general and do not enable "too many" reductions, we introduce functions that map mixed terms to terms over the original signature of the CTRS in question.

We define two basic approaches of projecting mixed terms in the transformed system back into corresponding original terms. The crucial idea is that when we consider a U-(sub)term $U^\delta(s_1, \ldots, s_n)$ in a given $\mathcal{R}'$-reduction we know that the root-symbol U^δ indicates that previously the introduction rule for $U^\delta\colon l \to r \Longleftarrow u_1 \to^* v_1, \ldots, u_n \to^* v_n$ must have been applied. Now, in order to get rid of U^δ, there are two natural ways of doing so: We can go back to the corresponding instance of the *lhs* l, or we anticipate the result by taking the corresponding instance of the *rhs* r. In both cases, the projection needs to recursively translate also U-subterms of the given term.

Definition 2.2 (translate backwards (tb))**.** Let $\mathcal{R} = (\mathcal{F}, R)$ be a normal 1-CTRS. Then the *translate backward* function $\mathsf{tb}\colon \mathcal{T} \to \mathcal{T}'$ is defined by

$$\mathsf{tb}(t) = \begin{cases} x & \text{if } t = x \in \mathcal{V} \\ f(\mathsf{tb}(t_1), \ldots, \mathsf{tb}(t_m)) & \text{if } t = f(t_1, \ldots, t_m) \text{ and } f \in \mathcal{F} \\ l\sigma & \text{if } t = U^\delta(v_1, v_2, \ldots, v_n, w_1, \ldots, w_k) \\ & \quad \text{and } \delta\colon l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n \end{cases}$$

where $\overrightarrow{Var(l)} = x_1, \ldots, x_k$ and σ is (recursively) defined as $x_i\sigma = \mathsf{tb}(w_i)$ for $1 \leq i \leq k$.

[3] Using $\overrightarrow{Var(t)}$ as sequence of the *set* of variables in t goes back to [15], whereas in [9, 8] the sequence is constructed from the *multiset* of variables in t. The former version appears to be generally preferable, because it is more abstract and avoids additional complications due to "non-synchronization effects".

Definition 2.3 (translate forward (tf)). Let $\mathcal{R} = (\mathcal{F}, R)$ be an normal 1-CTRS. Then the *translate forward* function $\mathsf{tf}\colon \mathcal{T} \to \mathcal{T}'$ is defined by

$$\mathsf{tf}(t) = \begin{cases} x & \text{if } t = x \in \mathcal{V} \\ f(\mathsf{tf}(t_1), \ldots, \mathsf{tf}(t_m)) & \text{if } t = f(t_1, \ldots, t_m) \text{ and } f \in \mathcal{F} \\ r\sigma & \text{if } t = U^\delta(v_1, v_2, \ldots, v_n, w_1, \ldots, w_k) \\ & \quad \text{and } \delta\colon l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n \end{cases}$$

where $\overrightarrow{Var(l)} = x_1, \ldots, x_k$ and σ is (recursively) defined as $x_i\sigma = \mathsf{tf}(w_i)$ for $1 \le i \le k$.

In this paper we focus on the property of *soundness* of unravelings which is dual to the (easier to obtain) property of completeness. An unraveling is said to be *complete* (for reductions) (or *simulation-complete*) if for all CTRSs $\mathcal{R}$, $s \to^*_{\mathcal{R}} t$ for $s, t \in \mathcal{T}$ implies $s \to^*_{\mathcal{R}'} t$. Furthermore, an unraveling is *sound* for reductions (or *simulation-sound*) if $s \to^*_{\mathcal{R}'} t$ implies $s \to^*_{\mathcal{R}} t$. Subsequently, we sometimes use a slightly more general notion of soundness by demanding that $s \to^*_{\mathcal{R}'} t$ (for $t \in \mathcal{T}'$) implies $s \to^*_{\mathcal{R}} \mathsf{tb}(t)$ resp. $\mathsf{tf}(t)$. This notion is indeed more general since $\mathsf{tb}(t) = \mathsf{tf}(t) = t$ whenever $t \in \mathcal{T}$ (i.e. t is an original term). Given a particular CTRS $\mathcal{R}$, we also say that the unraveling is complete (sound) for $\mathcal{R}$ or, slightly abusing terminology, that $\mathcal{R}'$ is complete (sound) w.r.t. $\mathcal{R}$. For a more thorough discussion of the terminology used for (preservation properties of) transformations we refer to [6].

3. (Un)Soundness for Normal 1-CTRSs

By carefully analyzing known counterexamples to soundness (of unravelings for normal 1-CTRSs) from the literature we first collect a couple of (mainly syntactic) properties whose absence may be viewed as tempting candidates for guaranteeing soundness (Subsection 3.1). We then show that some of them are not really essential for the unsoundness phenomenon.

3.1. Known and New Counterexamples

First of all, as observed in [6], there is a simple source of unsoundness in unravelings (as well as in most other transformations) which is due to an "optimized" version of unraveling as it is used in several papers. The underlying idea for this "optimization" is that when starting a conditional rule application via an introduction step, not all variable bindings of the *lhs* (instance) are stored in the corresponding U-term introduced, but only those that are needed to eventually produce the final *rhs* (instance), provided all conditions are satisfied. This motivates the definition of U_{opt} as follows: Transform

$$\delta\colon l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n$$

into

$$\begin{array}{rcll} l & \to & U^\delta(s_1, \ldots, s_n, \overrightarrow{Var(r)}) & \quad (\textit{introduction rule}) \\ U^\delta(t_1, \ldots, t_n, \overrightarrow{Var(r)}) & \to & r & \quad (\textit{elimination rule}) \end{array}$$

Given $\mathcal{R}$, let us denote the resulting system as $\mathcal{R}'_{opt}$. Then it is easy to see that simulating $\mathcal{R}$ (on $\mathcal{T}$) is indeed possible via $\mathcal{R}'_{opt}$, i.e., $\mathcal{R}'_{opt}$ is (simulation) complete (w.r.t. $\mathcal{R}$). However, concerning soundness (and consequently also e.g. completeness w.r.t. termination) there is a problem (due to non-left-linear rules in $\mathcal{R}$).

Example 3.1. When we unravel

$$\mathcal{R} = \left\{ \begin{array}{rcl} f(x) \to a & \Longleftarrow & b \to^* c \\ g(x,x) \to d & & \end{array} \right\}$$

with U_{opt} into

$$\mathcal{R}'_{opt} = \left\{ \begin{array}{c} f(x) \to U(b) \\ U(c) \to a \\ g(x,x) \to d \end{array} \right\}$$

we get $g(f(a), f(b)) \to^*_{\mathcal{R}_{opt}} g(U(b), U(b)) \to_{\mathcal{R}_{opt}} d$, but obviously $g(f(a), f(b)) \not\to^*_{\mathcal{R}} d$, because $f(t)$ is $\mathcal{R}$-irreducible for every $\mathcal{R}$-irreducible $t \in \mathcal{T}$.

If we now add the rule $d \to g(f(a), f(b))$ to $\mathcal{R}$, the resulting system is still terminating, but its unraveled version becomes non-terminating. ■

This subtle flaw of "optimized" transformations (caused by omitting certain seemingly unnecessary variable bindings) as for U_{opt} above has been overlooked in various papers on transformations (cf. e.g. [2], [8]).[4] But even if we exclude such "optimizations" and insist on keeping all variable bindings in introduction steps (as in U), unraveled systems are in general not sound, as discovered by Marchiori in his pioneering paper [9].[5] This is a striking fact that — at least at first glance – is rather counterintuitive!

The following is a slightly simplified version of the basic ingenious counterexample of Marchiori [8, Example 4.3], similar to [6, Example 1].

Example 3.2. Unraveling of $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ with

$$\underbrace{\begin{array}{ccccc} a & \to & c & \to & e \\ & \nearrow\!\!\!\!\searrow & & \searrow & \\ b & \to & d & \to & k \end{array} \quad \begin{array}{c} h(x,x) \to g(x,x,f(k)) \\ g(d,x,x) \to A \end{array}}_{\mathcal{R}_1} \qquad \underbrace{f(x) \to x \Longleftarrow x \to^* e}_{\mathcal{R}_2}$$

yields $\mathcal{R}' = \mathcal{R}_1 \cup \mathcal{R}'_2$ with

$$\underbrace{f(x) \to U(x,x) \quad U(e,x) \to x}_{\mathcal{R}'_2}$$

In $\mathcal{R}'$ we get

$$\begin{array}{rcllcl} & & h(f(a), f(b)) & \to^+ & h(U(c,d), U(c,d)) & \to & g(U(c,d), U(c,d), f(k)) \\ \to^+ & & g(d, U(c,d), f(k)) & \to^+ & g(d, U(k,k), U(k,k)) & \to & A \end{array}$$

However, in $\mathcal{R}$ we do not have $h(f(a), f(b)) \to^* A$, since otherwise this would imply

$$h(f(a), f(b)) \to^* h(s,s) \to g(s,s,f(k)) \to^* g(d,t,t) \to^* A$$

for some s, t satisfying (1) $f(a) \to^* s$, $f(b) \to^* s$, (2) $s \to^* d$, and (3) $s \to^* t$, $f(k) \to^* t$.

But (1) and (2) imply $s = d$, hence $t = d$ or $t = k$. However, by (3), $f(k) \to^* t$ is neither possible for $t = d$ nor for $t = k$. ■

[4] Also in [12] a similar optimized transformation is used. Although the results presented in [12] do not contradict examples like Example 3.1 above, the general problem with such "optimized" transformations remains hidden, cf. [12, counterex. R_4,p. 9].

[5] More precisely, the details are only included in the extended technical report version [8] of [9].

Inspection of Example 3.2 reveals that it has numerous properties that one might be tempted to conjecture to be essential for the counterexample property.

Observation 3.3. The system $\mathcal{R}$ of Example 3.2 enjoys the following (mostly syntactical) properties: It is non-left-linear ($\neg\mathsf{LL}$), non-confluent ($\neg\mathsf{CR}$), erasing, i.e. not non-erasing ($\neg\mathsf{NE}$), non-right-linear ($\neg\mathsf{RL}$), not a constructor system ($\neg\mathsf{CS}$), not an overlay system ($\neg\mathsf{OS}$), overlapping, i.e. not non-overlapping ($\neg\mathsf{NO}$) and collapsing, i.e. not non-collapsing ($\neg\mathsf{NCOL}$).

We will now investigate whether each of these properties is essential for unsoundness or not.

Proposition 3.4. *None of the properties of being*

- *not a constructor system ($\neg\mathsf{CS}$)*
- *not an overlay system ($\neg\mathsf{OS}$)*
- *collapsing ($\neg\mathsf{NCOL}$)*
- *non-right-linear ($\neg\mathsf{RL}$)*

is essential for unsoundness of unravelings.

Proof. Cf. Example 3.5 ∎

Example 3.5. Unraveling of $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ with

$$\underbrace{\begin{array}{ccccc} a & \to & c & \to & e \\ & \searrow\!\!\!\!\nearrow & & \searrow & \\ & & & & k \\ & \nearrow\!\!\!\!\searrow & & \nearrow & \\ b & \to & d & \to & l \end{array} \qquad g(x,x) \to A}_{\mathcal{R}_1} \qquad \underbrace{\begin{array}{ll} f(x) \to m(x) & \Longleftarrow x \to^* e \\ h(x,x) \to g(x, f(k)) & \Longleftarrow x \to^* m(l) \end{array}}_{\mathcal{R}_2}$$

(i.e. $a \to c$, $a \to d$, $b \to c$, $b \to d$, $c \to e$, $c \to k$, $d \to k$, $d \to l$)

yields $\mathcal{R}' = \mathcal{R}_1 \cup \mathcal{R}_2'$ with

$$\underbrace{\begin{array}{ll} f(x) \to U^f(x,x) & h(x,x) \to U^h(x,x) \\ U^f(e,x) \to m(x) & U^h(m(l),x) \to g(x, f(k))\,. \end{array}}_{\mathcal{R}_2'}$$

In $\mathcal{R}'$ we have

$$\begin{array}{rlcl} & h(f(a), f(b)) \to^+ h(U^f(c,d), U^f(c,d)) & \to & U^h(U^f(c,d), U^f(c,d)) \\ \to^+ & U^h(m(l), U^f(c,d)) \to g(U^f(c,d), f(k)) & \to^+ & g(U^f(k,k), U^f(k,k)) \to A \end{array}$$

However, in $\mathcal{R}$ we do not have $h(f(a), f(b)) \to^* A$, analogously to the reasoning in Example 3.2.

Proposition 3.6. *The property of being overlapping is not essential for unsoundness of unravelings.*

Proof. The non-confluent overlapping part of the Examples 3.2 and 3.5 can easily be changed into a non-overlapping (but still non-confluent) sub-system such that the counterexample property is preserved by using rules of the shape $a(x,x) \to c(p,p)$ and $a(x,i(x)) \to d(p,p)$ to simulate a divergence $c \leftarrow a \to d$. Additionally, a rule $p \to i(p)$ is added. ∎

3.2. Sufficient Criteria for Soundness

In this section we will prove that each of the remaining properties of the CTRS of Example 3.2, namely being non-left-linear, non-confluent and erasing (i.e. not non-erasing) is indeed crucial for the counterexample, thus yielding corresponding soundness criteria.

For the case of left-linearity this has already been proved by Marchiori in [8].

Theorem 3.7 (left-linearity is sufficient ([8], cf. also [15])). *Left-linearity of $\mathcal{R}$ is sufficient for soundness of $\mathcal{R}'$.*

In the following we establish that confluence and non-erasingness of a CTRS $\mathcal{R}$ are sufficient to deduce that the unraveling of Definition 2.1 is sound w.r.t. $\mathcal{R}$. Moreover, we generalize the soundness result for left-linear systems by demanding only weak left-linearity (see Definition 3.22 below) instead of left-linearity.

3.2.1. *Confluence.*

An important property of unravelings is that variables may be duplicated when U-symbols are introduced. For instance in Example 3.2 such a duplication occurs in the rule $f(x) \to U(x,x)$. Thus, in an $\mathcal{R}'$-reduction after this rule is applied, the instantiated variables could be reduced to different terms. In Example 3.2 this happens when $U(a,a)$ is reduced to $U(c,d)$.

However, when transforming a term like $U(c,d)$ into a term from $\mathcal{T}$ for instance using tb either c or d is selected as instantiation of the single variable of the left-hand side of the corresponding conditional rewrite rule. In case of tb we would get $\mathsf{tb}(U(c,d)) = f(d)$. Regarding soundness this is problematic in general, since $U(c,d) \to^+_{\mathcal{R}'} d$ but $\mathsf{tb}(U(c,d)) = f(d) \not\to_{\mathcal{R}} d = \mathsf{tb}(d)$. The particular problem here is that $d \not\to^*_{\mathcal{R}} e$ and thus the conditional rule is not applicable to $f(d)$. Non-confluence, i.e. $d \leftarrow^+_{\mathcal{R}} a \to^+_{\mathcal{R}} e$ but d and e are not joinable, is crucial for this problem.

If $\mathcal{R}$ is confluent and $U(u,v)$ (with $u, v \in \mathcal{T}$) appears as redex w.r.t. a U-elimination rule in a $\mathcal{R}'$-reduction sequence starting from an original term (provided that U has been introduced by a rule $l \to U(x,x)$), we can prove that $v \to^*_{\mathcal{R}} u$ holds. This is achieved by showing that u and v have a common ancestor in $\mathcal{T}$ and since u is a ground normal form, confluence of $\mathcal{R}$ implies $v \to^*_{\mathcal{R}} u$.

First we prove an auxiliary lemma basically stating a kind of monotony under $\mathcal{T}'$-contexts of $\to_{\mathcal{R}}$ when tb is applied.

Lemma 3.8 (monotony property of tb). *Let $\mathcal{R} = (\mathcal{F}, R)$ be a 1-CTRS. If $u \to_{p,\mathcal{R}'} v$ for terms $u, v \in \mathcal{T}'$ and $\mathsf{tb}(u|_p) \to^{\leq 1}_{\mathcal{R}} \mathsf{tb}(v|_p)$, then $\mathsf{tb}(u|_q) \twoheadrightarrow\!\!\!\!\!\!\parallel_{\mathcal{R}} \mathsf{tb}(v|_{q'})$ for all $q \in Pos(u)$ and all descendants q' of q in v.*

Proof (sketch). For the interesting case where $q \leq p$ we use induction on the size of p' determined by $q.p' = p$. ∎

The next lemma is the technical key result for the proof of Theorem 3.12 below. It states that in an $\mathcal{R}'$-reduction sequence D starting from an original term, for every redex u and its (one-step) reductum v appearing in D we have $\mathsf{tb}(u) \to^{\leq 1}_{\mathcal{R}} \mathsf{tb}(v)$.

Lemma 3.9 (technical key result for confluent systems)**.** *Let $\mathcal{R} = (\mathcal{F}, R)$ be a* confluent *normal 1-*CTRS *and let $D\colon u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \ldots \to_{p_{n-1},\mathcal{R}'} u_n$ be a reduction sequence where $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 < i \leq n$. Then, $\mathsf{tb}(u_i|_{p_i}) \to_{\mathcal{R}}^{\leq 1} \mathsf{tb}(u_{i+1}|_{p_i})$ for all $1 \leq i < n$.*

Proof (sketch). Proof by induction on the length of D and case distinction on the rule applied in the last step of D. The interesting case is where this last step is a U-elimination step. There, we get for every condition $s_i \to^* t_i$ of the corresponding conditional rule α that $\mathsf{tb}(s_i\sigma) \to_{\mathcal{R}}^* \mathsf{tb}(t_i)$ and $\mathsf{tb}(s_i\sigma) \to_{\mathcal{R}}^* \mathsf{tb}(s_i\tau)$ holds, where τ is the matcher used in the last step of D and σ the matcher used in the corresponding U-introduction step of α, according to the induction hypothesis. Then, confluence of $\mathcal{R}$ yields $\mathsf{tb}(s_i\tau) \to_{\mathcal{R}}^* t_i$ since $\mathsf{tb}(t_i) = t_i$ and t_i is a (ground $\mathcal{R}_u$-) normal form. Hence, α is applicable to $\mathsf{tb}(u_{n-1}|_{p_{n-1}})$ and we get $\mathsf{tb}(u_{n-1}|_{p_{n-1}}) \to_{\mathcal{R}} \mathsf{tb}(u_n|_{p_{n-1}})$. ∎

In Lemma 3.9 the confluence assumption cannot be dropped.

Example 3.10. Consider the following normal 1-CTRS $\mathcal{R}$.

$$a \to b \qquad a \to c \qquad f(x) \to x \Leftarrow x \to^* b$$

$\mathcal{R}$ is not confluent since b and c are not joinable. Consider the $\mathcal{R}'$-reduction sequence $f(a) \to_{\mathcal{R}'} U(a,a) \to_{\mathcal{R}'}^{+} U(b,c) \to_{\mathcal{R}'} c$ and the term $U(b,c)$. In the proof of Lemma 3.9 we showed that b and c must have a common ancestor. However, while in the proof we used this fact to deduce that they also have a common descendant and further that this descendant must be b, in the example this conclusion is wrong because of non-confluence of $\mathcal{R}$. Indeed, Lemma 3.9 does not hold for this example, since $\mathsf{tb}(U(b,c)) = f(c) \not\to_{\mathcal{R}} c = \mathsf{tb}(c)$.

Lemma 3.11 (projecting reductions issuing from original terms)**.** *Let $\mathcal{R}$ be confluent. Then for every $\mathcal{R}'$-reduction $u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \ldots \to_{p_{n-1},\mathcal{R}'} u_n$ with $u_1 \in \mathcal{T}$ we have $u_1 = \mathsf{tb}(u_1) \mathrel{+\!\!\!\!\!\!\twoheadrightarrow}_{\mathcal{R}} \mathsf{tb}(u_2) \mathrel{+\!\!\!\!\!\!\twoheadrightarrow}_{\mathcal{R}} \ldots \mathrel{+\!\!\!\!\!\!\twoheadrightarrow}_{\mathcal{R}} \mathsf{tb}(u_n)$.*

Proof. For every redex $u_j|_{p_j}$ and corresponding reductum $u_{j+1}|_{p_j}$ $(1 \leq j < n)$ we have $\mathsf{tb}(u_j|_{p_j}) \to_{\mathcal{R}}^{\leq 1} \mathsf{tb}(u_{j+1}|_{p_j})$ because of Lemma 3.9. This implies $\mathsf{tb}(u_j) \mathrel{+\!\!\!\!\!\!\twoheadrightarrow}_{\mathcal{R}} \mathsf{tb}(u_{j+1})$ according to Lemma 3.8 (with $q = q' = \epsilon$). ∎

As corollary we obtain the following result.

Theorem 3.12 (confluence is sufficient)**.** *Confluence of $\mathcal{R}$ is sufficient for soundness of $\mathcal{R}'$.*

Proof. Straightforward using Lemma 3.11. ∎

3.2.2. *Non-Erasingness.*

In Example 3.2 the $\mathcal{R}'$-reduction that is a witness for unsoundness contains U-(sub)-terms that are not reducible to original terms, since the U-symbol cannot be eliminated (e.g. the term $U(k,k)$). Hence, since the final term A of the reduction is an original term, these terms must be erased.

When considering a non-erasing CTRS $\mathcal{R}$ (and thus a non-erasing $\mathcal{R}'$), every U-symbol in every (finite) $\mathcal{R}'$ reduction sequence D ending in a term from $\mathcal{T}$ must be properly eliminated. This fact motivates and justifies the use of tf when simulating $\mathcal{R}'$-reductions in $\mathcal{R}$, as whenever some U-term is encountered in D it will eventually be eliminated in D and this elimination is anticipated when applying tf.

The following lemma is dual to Lemma 3.8 in that tf instead of tb is used for transforming terms from $\mathcal{T}'$ into terms from $\mathcal{T}$.

Lemma 3.13 (monotony property of tf). *Let $\mathcal{R} = (\mathcal{F}, R)$ be a 1-CTRS. If $u \rightarrow_{p,\mathcal{R}'} v$ for $u, v \in \mathcal{T}'$ and $\mathsf{tf}(u|_p) \rightarrow^{\leq 1}_{\mathcal{R}} \mathsf{tf}(v|_p)$, then $\mathsf{tf}(u|_q) \twoheadrightarrow\!\!\!\!\!\!\shortparallel\;_{\mathcal{R}} \mathsf{tf}(v|_{q'})$ for all $q \in Pos(u)$ and all descendants q' of q in v.*

Proof (sketch). The proof is analogous to the one of Lemma 3.8. For the interesting case where $q \leq p$ we use induction on the size of p' determined by $q.p' = p$. ∎

The next lemma is the technical key result for the proof of Theorem 3.16 below. It is dual to Lemma 3.9 in that tf is used instead of tb.

Lemma 3.14 (technical key result for non-erasing systems). *Let $\mathcal{R} = (\mathcal{F}, R)$ be a* non-erasing *normal 1-CTRS and let $D : u_1 \rightarrow_{p_1,\mathcal{R}'} u_2 \rightarrow_{p_2,\mathcal{R}'} \ldots \rightarrow_{p_{n-1},\mathcal{R}'} s_n$ be a reduction sequence where $u_n \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 \leq i < n$. Then, $\mathsf{tf}(u_i|_{p_i}) \rightarrow^{\leq 1}_{\mathcal{R}} \mathsf{tf}(u_{i+1}|_{p_i})$ for $1 \leq i < n$.*

Proof (sketch). Proof by induction on the length of D and case distinction on the rule applied in the first step of D. The interesting case is where this first step is a U-introduction step. Since $\mathcal{R}'$ is non-erasing, the introduced U-symbol is eventually eliminated in D and hence by the induction hypothesis and Lemma 3.13 we get $\mathsf{tf}(s_i\sigma) \rightarrow^*_{\mathcal{R}} t_i$ for all conditions of the conditional rule corresponding to the introduced U-symbol. Hence $\mathsf{tf}(u_1|_{p_1}) \rightarrow_{\mathcal{R}} \mathsf{tf}(u_2|_{p_1})$. ∎

Finally, we can prove soundness of unravelings for non-erasing normal 1-CTRSs.

Lemma 3.15 (projecting reductions issuing from original term). *Let $\mathcal{R}$ be non-erasing. Then for every $\mathcal{R}'$-reduction $u_1 \rightarrow_{p_1,\mathcal{R}'} u_2 \rightarrow_{p_2,\mathcal{R}'} \ldots \rightarrow_{p_{n-1},\mathcal{R}'} u_n$ with $u_n \in \mathcal{T}$ we have $\mathsf{tf}(u_1) \twoheadrightarrow\!\!\!\!\!\!\shortparallel\;_{\mathcal{R}} \mathsf{tf}(u_2) \twoheadrightarrow\!\!\!\!\!\!\shortparallel\;_{\mathcal{R}} \ldots \twoheadrightarrow\!\!\!\!\!\!\shortparallel\;_{\mathcal{R}} \mathsf{tf}(u_{n-1}) \twoheadrightarrow\!\!\!\!\!\!\shortparallel\;_{\mathcal{R}} \mathsf{tf}(u_n) = u_n$.*

Proof. For every redex $u_j|_{p_j}$ and corresponding reductum $u_{j+1}|_{p_j}$ $(1 \leq j < n)$ we have $\mathsf{tf}(u_j|_{p_j}) \rightarrow^{\leq 1}_{\mathcal{R}} \mathsf{tf}(u_{j+1}|_{p_j})$ because of Lemma 3.14. This implies $\mathsf{tf}(u_j) \twoheadrightarrow\!\!\!\!\!\!\shortparallel\;_{\mathcal{R}} \mathsf{tf}(u_{j+1})$ according to Lemma 3.13 (with $q = q' = \epsilon$). ∎

Theorem 3.16 (non-erasingness is sufficient). *Non-erasingness of $\mathcal{R}$ is sufficient for soundness of $\mathcal{R}'$.*

Proof. Straightforward using Lemma 3.15. ∎

3.2.3. *Right-Linearity Revisited.*

Next we reconsider right-linearity. In Example 3.5 we have shown that non-right-linearity of $\mathcal{R}$ is not essential for unsoundness. However, in this example the unraveled system $\mathcal{R}'$ becomes non-right-linear. This property of $\mathcal{R}'$ is crucial for Example 3.5 (as we will see). Yet, demanding that $\mathcal{R}'$ is right-linear is a severe restriction, since right-linearity of $\mathcal{R}'$ implies that $\mathcal{R}$ contains only ground conditions. To see this consider some conditional rule $l \rightarrow r \Leftarrow s \rightarrow t$, such that $x \in Var(s)$. Since we consider 1-CTRSs this implies $x \in Var(l)$ and hence the unraveled system contains a non-right-linear rule $l \rightarrow U(s, x)$.

It turns out that for CTRSs $\mathcal{R}$ having only ground conditions (GC), $\mathcal{R}'$ is sound even if $\mathcal{R}$ is not right-linear.

Theorem 3.17 (GC is sufficient for soundness)**.** *If $\mathcal{R}$ has only ground conditions, then $\mathcal{R}'$ is sound (w.r.t. $\mathcal{R}$).*

Proof (sketch). The proof is basically analogous to the proof of soundness for confluent CTRSs. There, confluence was (exclusively) needed to show that $t_i \leftarrow^*_{\mathcal{R}} \mathsf{tb}(s_i\sigma) \rightarrow^*_{\mathcal{R}} \mathsf{tb}(s_i\tau)$ implies $\mathsf{tb}(s_i\tau) \rightarrow^*_{\mathcal{R}} t_i$ for conditions $s_i \rightarrow^* t_i$ of some conditional rule and certain substitutions τ and σ (cf. the proof of Lemma 3.9). However, for CTRSs with ground conditions this is trivial since $s_i\sigma = s_i\tau$ for all substitutions σ and τ and thus $\mathsf{tb}(s_i\sigma) = \mathsf{tb}(s_i\tau)$. ■

Of course, systems with only ground conditions are of limited practical use (and could in principle, though not necessarily effectively, be replaced by equivalent unconditional systems).

3.2.4. *Normal Form Property.*

Reconsidering the sufficiency of confluence of $\mathcal{R}$ for soundness (Theorem 3.12), we can get another slightly more general criterion.

Regarding confluence properties, the following proper implications (for TRSs and also for ARSs) are well-known (cf. e.g. [15]):

$$(*) \quad \mathsf{CR} \Longrightarrow \mathsf{NF} \Longrightarrow \mathsf{UN} \Longrightarrow \mathsf{UN}^{\rightarrow}\,.$$

In the proof of Theorem 3.12, what is actually needed, is not full confluence, but only the property

$$(+) \quad t\ {}^*\!\!\leftarrow_{\mathcal{R}} s \rightarrow^* u \in \mathsf{NF}(\mathcal{R}) \Longrightarrow t \rightarrow^* u\,.$$

Proposition 3.18. *Property (+) is equivalent to* NF*.*

Proof. Straightforward. ■

Consequently we can generalize Theorem 3.12 slightly as follows.

Theorem 3.19 (NF is sufficient)**.** *The normal form property (*NF*) of $\mathcal{R}$ is sufficient for soundness of $\mathcal{R}'$.*

Regarding the above proper implications (*) and Theorem 3.19, an obvious question is whether UN or $\mathsf{UN}^{\rightarrow}$, respectively, is sufficient for soundness.

Proposition 3.20 (UN and $\mathsf{UN}^{\rightarrow}$ are not sufficient for soundness)**.** UN *and* $\mathsf{UN}^{\rightarrow}$ *are not sufficient for soundness.*

Proof. Cf. Example 3.21. ■

Example 3.21 (Example 3.2 continued)**.** Consider the system $\widehat{\mathcal{R}}$ obtained from $\mathcal{R}$ as in Example 3.2 by adding the additional unconditional rule $k \rightarrow k$. Then it is easy to verify that $\widehat{\mathcal{R}}$ is not NF, but UN and $\mathsf{UN}^{\rightarrow}$. Moreover, $\widehat{\mathcal{R}}'$ is still unsound w.r.t. $\widehat{\mathcal{R}}$.

3.2.5. *Left-Linearity Revisited.*

It is well-known that left-linear join (1-)CTRSs can be simulated by left-linear normal (1-)CTRSs extended by an additional rule like $eq(x,x) \to tt$ (yielding $\mathcal{R}_{eq}$), via encoding join conditions $u_i \downarrow_{\mathcal{R}} v_i$ as $eq(u_i, v_i) \to^*_{\mathcal{R}_{eq}} tt$. Hence, it would be interesting to know whether – regarding left-linearity of $\mathcal{R}$ as sufficient criterion for soundness (Theorem 3.7) – this class could be extended slightly so as to cover also left-linear systems extended by (non-left-linear) "eq-like" rules. This is indeed the case as we will show next.

Definition 3.22 (weak left-linearity)**.** A normal 1-CTRS is said to be *weakly left-linear* if every rule $l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n$ of $\mathcal{R}$ is either left-linear or, if not, is unconditional and every non-linear variable in l does not occur at all in r.[6]

In particular, extending (not necessarily disjointly concerning the signature) a left-linear normal 1-CTRS by $eq(x,x) \to tt$ yields a weakly left-linear system.

Before proving that weak-left-linearity of $\mathcal{R}'$ is indeed sufficient for soundness of unravelings, we state two observations regarding the preservation of weak left-linearity under unravelings and the existence and uniqueness of one-step ancestors of U-terms in reductions w.r.t. weakly left-linear systems $\mathcal{R}'$.

Observation 3.23. $\mathcal{R}$ is weakly left-linear iff $\mathcal{R}'$ is so.

Observation 3.24. Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS. If $u \to_{p,\mathcal{R}'} v$, then every U-rooted subterm position of v has exactly one one-step ancestor in u.

Proof. For all normal 1-CTRSs every U-(sub)-term has at least one one-step ancestor (in an $\mathcal{R}'$-reduction), because U-symbols do not occur strictly below the root of *rhs*'s of rules in $\mathcal{R}'$. Weak left-linearity of $\mathcal{R}$ implies weak left-linearity of $\mathcal{R}'$ and thus in every $\mathcal{R}'$-reduction every term has at most one one-step ancestor. ∎

Observation 3.24 motivates the definition of a function tb_D w.r.t. to a $\mathcal{R}'$-reduction sequence D, starting from an original term, which basically transforms terms from $\mathcal{T}'$ into terms from $\mathcal{T}$. Since we can trace a U-(sub)term uniquely backwards in D (uniqueness is due to Observation 3.24), the idea is that we can find the first (when traced backwards) non-U-rooted ancestor of the U-term (i.e., the one appearing in the term of D with the highest index) and thus replace the U-(sub)term by this ancestor.

Definition 3.25 (tb_D)**.** Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS and let $D : u_1 \to_{\mathcal{R}'} u_2 \to_{\mathcal{R}'} \ldots, \to_{\mathcal{R}'} u_n$ be a reduction sequence with $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 < i \leq n$. We define the (partial) function $\mathsf{tb}_D \colon \{1, \ldots, n\} \times \mathbb{N}_+^* \to \mathcal{T}$, i.e. from pairs (i,p), where i is an index and p is a position, as

$$\mathsf{tb}_D(i,p) = \begin{cases} \text{undefined} & \text{if } p \notin Pos(u_i) \\ x & \text{if } u_i|_p = x \in \mathcal{V} \\ f(\mathsf{tb}_D(i,p.1), \ldots, \mathsf{tb}_D(i,p.l)) & \text{if } u_i|_p = f(t_1, \ldots, t_l) \text{ and } f \in \mathcal{F} \\ \mathsf{tb}_D(i-1,p') & \text{if } root(u_i|_p) \in \mathcal{F}' \setminus \mathcal{F},\ i > 1 \text{ and } u_{i-1}|_{p'} \text{ is the} \\ & \quad \text{unique one-step ancestor of } u_i|_p\,. \end{cases}$$

Note that pairs (i,p) are supposed to determine a subterm occurrence at position p in the ith term of D. Hence, tb_D is undefined if the pair does not determine such a term, i.e. if $p \notin Pos(u_i)$.

[6]Note that this definition also covers the case of TRSs.

Example 3.26. Let $\mathcal{R}$ be as in Example 3.2 and consider the $\mathcal{R}'$-derivation $D\colon u_1 = f(a) \to_{\mathcal{R}'} U(a,a) \to_{\mathcal{R}'} U(a,d) \to_{\mathcal{R}'} U(c,d) = u_4$. Then we have $\mathsf{tb}(U(c,d)) = f(d)$, but $\mathsf{tb}_D(4,\epsilon) = f(a)$ (here, $u_4 = U(c,d)|_\epsilon = U(c,d)$). Note that the backtranslation tb_D goes back further than tb. For instance, we have $\mathsf{tb}_D(U(c,d) \to^*_{\mathcal{R}'} d$, but $\mathsf{tb}(U(c,d) \not\to^*_{\mathcal{R}'} d$.

The following lemma roughly states that whether the tb_D-version of some (sub)term is reachable in $\mathcal{R}$ by the tb_D-version of its ancestor depends only on whether the tb_D-version of the reductum is reachable by the tb_D-version of the redex in the corresponding step.

Lemma 3.27 (monotony property of tb_D)**.** *Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS and let $D\colon u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \cdots \to_{p_{n-1},\mathcal{R}'} u_n$ be an $\mathcal{R}'$-reduction sequence with $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 < i \leq n$. If $\mathsf{tb}_D(i,p_i) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1,p_i)$ for every $1 \leq i \leq n-1$, then $\mathsf{tb}_D(i,p) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1,p')$ for every $1 \leq i \leq n-1$, every $p \in Pos(u_i)$ and every descendant $u_{i+1}|_{p'}$ of $u_i|_p$.*

Proof (sketch). For the interesting case where $p \leq p_i$ we use induction on the size of $\overline{p}$ determined by $p.\overline{p} = p_i$. ∎

In Lemma 3.28 below we prove a restricted monotony property of tb_D.

Lemma 3.28 (extraction of tb_D in U-rooted terms)**.** *Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS and $D\colon u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \cdots \to_{p_{n-1},\mathcal{R}'} u_n$ be an $\mathcal{R}'$-reduction sequence with $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 < i \leq n$. If $\mathsf{tb}_D(i,p_i) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1,p_i)$ for every $1 \leq i \leq n-1$, $u_k|_p = U^\alpha(v_1,\ldots,v_{m_1},x_1,\ldots,x_{m2})\tau$, $\alpha = l \to r \Leftarrow s_1 \to^* t_1,\ldots,s_{m_1} \to t_{m_1}$ and $\mathsf{tb}_D(k,p) = l\sigma$, then $s_i\sigma \to^*_{\mathcal{R}} \mathsf{tb}_D(k,p.i)$ for all $1 \leq i \leq m_1$ and $x_i\sigma \to^*_{\mathcal{R}} \mathsf{tb}_D(k,p.(m_1+i))$ for all $1 \leq i \leq m_2$.*

Proof (sketch). Proof by induction on k and using Lemma 3.27. ∎

The next lemma shows that the backtranslation of tb is intuitively not "as far back" as the one of tb_D by stating that $\mathsf{tb}_D(i,p) \to^*_{\mathcal{R}} \mathsf{tb}(u_i|_p)$ for certain $\mathcal{R}'$-reductions D.

Lemma 3.29 (tb_D to tb)**.** *Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS and let $D\colon u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \cdots \to_{p_{n-1},\mathcal{R}'} u_n$ be a $\mathcal{R}'$-reduction sequence with $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 < i \leq n$. If $\mathsf{tb}_D(i,p_i) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1,p_i)$ for every $1 \leq i \leq n-1$, then $\mathsf{tb}_D(j,p) \to^*_{\mathcal{R}} \mathsf{tb}(u_j|_p)$ for all $1 \leq j \leq n$ and all $p \in Pos(u_j)$.*

Proof (sketch). Proof by induction on the term depth of $u_j|_p$ and using Lemma 3.28. ∎

The following lemma is the technical key result for soundness in the weakly left-linear case. It states that in every $\mathcal{R}'$-reduction D we have $\mathsf{tb}_D(i,p) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1,p)$, if $u_i|_p$ is the redex contracted in D.

Lemma 3.30 (technical key result for weakly left-linear systems)**.** *Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS and let $D\colon u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \cdots \to_{p_{n-1},\mathcal{R}'} u_n$ be a $\mathcal{R}'$-reduction sequence with $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 < i \leq n$, then $\mathsf{tb}_D(i,p_i) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1,p_i)$ for all $1 \leq i < n$.*

Proof (sketch). Proof by induction on the length of D and case distinction over the applied rule in the last reduction step of D. There are two interesting cases. First, if the rule is an unconditional non-left-linear rule $l \to r$, this rule might not be applicable to $\mathsf{tb}_D(n-1,p_{n-1})$ since $u_{n-1}|_q = u_{n-1}|_{q'} \not\Rightarrow \mathsf{tb}_D(n-1,q) = \mathsf{tb}_D(n-1,q')$. However, by Lemma 3.29 we get $\mathsf{tb}_D(n-1,q) \to^*_{\mathcal{R}} \mathsf{tb}(u_{n-1}|_q)$ and $\mathsf{tb}_D(n-1,q') \to^*_{\mathcal{R}} \mathsf{tb}(u_{n-1}|_{q'})$. Hence, $\mathsf{tb}_D(n-1,p_{n-1}.q) \downarrow_{\mathcal{R}}$

$\mathsf{tb}_D(n-1, p_{n-1}.q')$ for all positions q, q' where $l|_q = l|_{q'} = x \in \mathcal{V}$. Moreover, these reductions do not effect the reductum after the rule is applied, since all non-linear variables are erased due to weak left-linearity of $\mathcal{R}$.

For the second interesting case where the last applied rule is a U-elimination rule we get $s_i\sigma \to^*_{\mathcal{R}} t_i$ according to Lemma 3.28 for every condition $s_i \to^* t_i$ of the conditional rewrite rule corresponding to the eliminated U-symbol, where σ is given by $\mathsf{tb}_D(n-1, p_{n-1}) = l\sigma$. Hence, this implies $\mathsf{tb}_D(n-1, p_{n-1}) \to^*_{\mathcal{R}} \mathsf{tb}_D(n, p_{n-1})$ by again applying Lemma 3.28. ∎

Weak left-linearity is crucial in Lemma 3.30 to ensure that non-left-linear rules are applicable in the tb_D-versions of redexes.

Example 3.31. Consider the weakly left-linear normal 1-CTRS $\mathcal{R}$ given by

$$eq(x,x) \to tt \qquad f(x) \to b \Leftarrow x \to^* b$$
$$a \to b$$

and the $\mathcal{R}'$-derivation

$$D\colon eq(f(a), f(b)) \to^+_{\mathcal{R}'} eq(U(a,a), U(b,b)) \to^+_{\mathcal{R}'} eq(b,b) \to_{\mathcal{R}'} tt\,.$$

Let $u_{n-1} = eq(b,b)$, then $\mathsf{tb}_D(n-1,\epsilon) = u_1 = eq(f(a), f(b))$ and $eq(f(a), f(b)) \not\to_{\mathcal{R}} tt$ (i.e., with one single $\mathcal{R}$-step). However, $f(a)$ and $f(b)$ are joinable (in general this is justified by Lemma 3.29) and reducing them is not problematic as the non-linear variable x is erased whenever the eq-rule is applied (this must in general be the case because of weak left-linearity of $\mathcal{R}$). Hence, we have $\mathsf{tb}_D(n-1,\epsilon) \to^*_{\mathcal{R}} tt = \mathsf{tb}_D(n,\epsilon)$.

The following lemma and theorem state the main soundness result for weakly left-linear normal 1-CTRSs.

Lemma 3.32. *Let $\mathcal{R}$ be a weakly left-linear normal 1-CTRS and let $D\colon u_1 \to_{p_1,\mathcal{R}'} u_2 \to_{p_2,\mathcal{R}'} \ldots \to_{p_{n-1},\mathcal{R}'} u_n$ be an $\mathcal{R}'$-reduction sequence with $u_1 \in \mathcal{T}$ and $u_i \in \mathcal{T}'$ for $1 \le i \le n$. Then, $u_1 = \mathsf{tb}_D(1,\epsilon) \to^*_{\mathcal{R}} \mathsf{tb}(u_n)$.*

Proof. Lemma 3.30 yields that $\mathsf{tb}_D(i, p_i) \to^*_{\mathcal{R}} \mathsf{tb}_D(i+1, p_i)$ for all $1 \le i < n$. Hence, Lemma 3.27 is applicable and its repeated application yields $\mathsf{tb}_D(1,\epsilon) \to^*_{\mathcal{R}} \mathsf{tb}_D(n,\epsilon)$. Finally, Lemma 3.29 yields $\mathsf{tb}_D(n,\epsilon) \to^*_{\mathcal{R}} \mathsf{tb}(u_n)$. ∎

Theorem 3.33. *Weak left-linearity of $\mathcal{R}$ is sufficient for soundness of $\mathcal{R}'$.*

Proof. Straightforward using Lemma 3.32. ∎

Obviously, Theorem 3.33 properly generalizes Theorem 3.7. Intuitively, the former result and its proof show that non-left-linearity due to "eq-like" rules is not problematic, since the effects of applying such a non-left-linear rule are only local (and do not cause complex sharing of equal subterms along longer derivations).

A nice consequence of Theorem 3.33 is that left-linear join 1-CTRSs $\mathcal{R}_j$ can be soundly unraveled (via the unraveling U for the case of normal 1-CTRSs) by first encoding $\mathcal{R}_j$ into a normal 1-CTRS $\mathcal{R}_n$ (in a many-sorted setting, by adding the rule $eq(x,x) \to tt$ to $\mathcal{R}_j$ where $eq\colon s \times s \to bool$ is a fresh binary function symbol of sort $bool$ and tt a fresh constant of sort $bool$, and all terms $s \in \mathcal{T}$ are considered as s-sorted, with $s \neq bool$, and by representing conditions $u_i \downarrow v_i$ as $eq(u_i, v_i) \to^* tt$) and a subsequent unraveling of $\mathcal{R}_n$ into $\mathcal{R}'_n$.

4. Discussion, Perspectives and Related Work

First let us summarize the results obtained. The table in Figure 1 lists the properties (of $\mathcal{R}$) investigated in the first row, indicates whether they are sufficient for soundness (of $\mathcal{R}'$) in the second row (+ means "Yes", − "No"), and gives references for the results in the last row.

LL	CS	OS	RL	NO	CR	NE	NF	GC	UN	$\mathsf{UN}^{\rightarrow}$	WLL
+	−	−	−	−	+	+	+	+	−	−	+
3.7 ([8, TH. 6.12])	3.4	3.4	3.4	3.6	3.12	3.16	3.16	3.17	3.20	3.20	3.33

Figure 1: Sufficiency of conditions for soundness of unravelings (of normal 1-CTRSs)

Due to the carefully designed modular proof structure of the obtained positive results and to the conceptually clear underlying ideas and the corresponding projection approaches (via tb, tf and tb_D) we expect that at least some of the results can be extended to other classes of CTRSs and to other transformations from CTRSs to TRSs. One case, for which this is indeed possible, concerns an alternative *sequential* version of unraveling normal 1-CTRSs. Here, the idea is that the conditions of a conditional rule are not processed simultaneously (by the unraveling), but sequentially, one at a time. This means, given the rule $\delta\colon l \to r \Longleftarrow s_1 \to^* t_1, \ldots, s_n \to t_n$, instead of one introduction rule $l \to U^\delta(s_1, \ldots, s_n, \overrightarrow{Var(l)})$ and one elimination rule $U^\delta(t_1, \ldots, t_n, \overrightarrow{Var(l)}) \to r$ we have one first introduction rule $l \to U_1^\delta(s_1, \overrightarrow{Var(l)})$, $n-2$ further intermediate "switch"-rules $U_i^\delta(t_i, \overrightarrow{Var(l)}) \to U_{i+1}^\delta(s_{i+1}, \overrightarrow{Var(l)})$, $1 \le i \le n-1$ (which act as elimination rules for U_i^δ and as introduction rules for U_{i+1}^δ) and a final elimination rule $U_{n-1}^\delta(t_n, \overrightarrow{Var(l)}) \to r$. All results (for U) presented in the paper actually also hold for this *sequential unraveling* U_{seq} as can be shown by a careful inspection and adaptation of the proofs.

The corresponding analysis of U_{seq} for normal 1-CTRSs provides the appropriate basis for dealing with the more general class of *deterministic* (oriented) 3-CTRSs where bindings for extra variables in the conditions and in right-hand side r of $l \to r \Longleftarrow s_1 \to^* t_1, \ldots, s_n \to^* t_n$ are "determined" by sequentially processing the conditions, i.e., $Var(s_i) \subseteq Var(l) \cup \bigcup_{1 \le j \le i-1} Var(t_j)$. But the details of this extension still need to be carefully worked out.

There are various open questions in the area. For instance, it remains unclear whether an even better (more precise) characterization of unsoundness exists, in the form of a general characterization result for unsoundness, similar to the one for non-modularity of termination (cf. e.g. [7, Theorem 7]), from which (most) known sufficient criteria for soundness follow.

Regarding related work, as far as we know left-linearity (of $\mathcal{R}$) was the only established sufficient criterion for soundness (of $\mathcal{R}'$), cf. [8, 9], [15, Chapter 7]. Compared to the proofs in these papers, we think that our proof of the more general Theorem 3.33 is in a sense more modular and less operational than these previous ones, and is also better suited for potential extensions.

Regarding more general classes of CTRSs (as compared to normal 1-CTRSs), the only works that we aware of, are [10] and [12]. However, in [10] there is only a claim ([10, Theorem 5.2], without any proof or proof sketch) stating soundness of (sequential) unravelings for *semilinear* DCTRSs, and in [12] the basic unraveling transformation used is a kind of optimized version analogous to U_{opt}, cf. Section 3.1 and Example 3.1, for which we

have argued that such an optimization is generally problematic from the point of view of soundness.

Acknowledgments

The authors are grateful to the anonymous referees for various useful hints and suggestions.

References

[1] F. Baader and T. Nipkow. *Term rewriting and All That*. Cambridge University Press, 1998.
[2] J. Bergstra and J. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.
[3] M. Bezem, J. Klop, and R. Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science 55. Cambridge University Press, Mar. 2003.
[4] N. Dershowitz and D. Plaisted. Logic programming cum applicative programming. In *Proc. 1985 Symposium on Logic Programming, Boston, Massachusetts, July 15-18, 1985*, pp. 54–66. IEEE, 1985.
[5] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(10):59–88, 2008.
[6] K. Gmeiner and B. Gramlich. Transformations of conditional rewrite systems revisited. In A. Corradini and U. Montanari, eds., *Recent Trends in Algebraic Development Techniques (WADT 2008) – Selected Papers*, LNCS 5486, pp. 166–186. Springer, 2009.
[7] B. Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5:131–158, 1994.
[8] M. Marchiori. Unravelings and ultra-properties. Technical Report 8 (37 pages, long version of [9]), University of Padova, Italy, 1995.
[9] M. Marchiori. Unravelings and ultra-properties. In M. Hanus and M. Rodríguez-Artalejo, eds., *Proc. 5th Int. Conf. on Algebraic and Logic Programming*, LNCS 1139, pp. 107–121. Springer, 1996.
[10] M. Marchiori. On deterministic conditional rewriting. Technical Report MIT LCS CSG Memo n. 405, MIT, Cambridge, MA, USA, Oct. 1997.
[11] N. Nishida, T. Mizutani, and M. Sakai. Transformation for refining unraveled conditional term rewriting systems. In S. Antoy, ed., *Final Proc. 6th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2006). Electr. Notes Theor. Comput. Sci. (ENTCS)*, 174(10), 2007.
[12] N. Nishida, M. Sakai, and T. Sakabe. On simulation-completeness of unraveling for conditional term rewriting systems. *IEICE Tech. Rep. SS2004-18*, 104(243):25–30, 2004. Revised version, 15 p., Dec. 2005.
[13] N. Nishida and M. Sakai. Completion after program inversion of injective functions. In A. MIddeldorp, ed., *Proc. 8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008)*, Castle of Hagenberg, Austria, 14 July 2008. *Electr. Notes Theor. Comput. Sci.*, 237:39–56, 2009.
[14] N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In J. Giesl, ed., *Proc. 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*, LNCS 346, pp. 264–278. Springer, Apr. 2005.
[15] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.
[16] G. Rosu. From conditional to unconditional rewriting. In J. L. Fiadeiro, P. D. Mosses, and F. Orejas, eds., *Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004), Revised selected papers*, LNCS 3423, pp. 218–233. Springer, 2004.
[17] F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *Journal of Logic and Algebraic Programming*, 2009. Revised selected papers of NWPT 2008, T. Uustalu and J. Vain, eds., to appear.
[18] Y. Toyama. Confluent term rewriting systems with membership conditions. In S. Kaplan and J.-P. Jouannaud, eds., *Proc. 1st Int. Workshop on Conditional Term Rewriting Systems, Orsay, France, July 8-10, 198*, LNCS 308, pp. 228–241. Springer, 1988.
[19] P. Viry. Elimination of conditions. *J. Symb. Comput.*, 28(3):381–401, 1999.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 135-150
http://rewriting.loria.fr/rta/

A PROOF CALCULUS WHICH REDUCES SYNTACTIC BUREAUCRACY

ALESSIO GUGLIELMI [1] AND TOM GUNDERSEN [2] AND MICHEL PARIGOT [3]

[1] University of Bath and LORIA & INRIA Nancy–Grand Est

[2] LIX & INRIA Saclay–Île-de-France

[3] Laboratoire PPS, UMR 7126, CNRS & Université Paris 7
E-mail address: parigot@pps.jussieu.fr

Abstract. In usual proof systems, like the sequent calculus, only a very limited way of combining proofs is available through the tree structure. We present in this paper a logic-independent proof calculus, where proofs can be freely composed by connectives, and prove its basic properties. The main advantage of this proof calculus is that it allows to avoid certain types of syntactic bureaucracy inherent to all usual proof systems, in particular the sequent calculus. Proofs in this system closely reflect their atomic flow, which traces the behaviour of atoms through structural rules. The general definition is illustrated by the standard deep-inference system for propositional logic, for which there are known rewriting techniques that achieve cut elimination based only on the information in atomic flows.

1. Introduction

One of the biggest challenges we are facing in structural proof theory, especially when looking at computational interpretations of proof systems, is syntactic dependency: formalisms impose irrelevant constraints, typically an arbitrary order between operations that are in principle independent from each other. The first explicit attempts to lower this syntactic dependency, called 'bureaucracy', date back to the eighties, with the concept of proof net for linear logic: proof nets are geometric traces of sequent-calculus proofs, which eliminate some syntactic constraints. Proof nets have been widely studied in the past two decades. Despite being powerful tools, they have two obvious limitations: 1) they apply directly only to specific logics, 2) they are not 'deductive' and rely on the sequent calculus for the deducing task.

1998 ACM Subject Classification: F.4.2.

Key words and phrases: Logic, Proof theory, Deep Inference, Flow graphs, Proof Systems, Open Deduction, Rewriting, Confluence, Termination.

© A. Guglielmi, T. Gundersen, and M. Parigot
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.135

A typical example of two proofs in the sequent calculus that are 'morally' the same but syntactically different, only because rules are applied in a different order, is the following:

$$\dfrac{\dfrac{\dfrac{\vdash a,\bar a \quad \vdash b,\bar b}{\vdash a\wedge b,\bar a,\bar b}\wedge}{\vdash a\wedge b,\bar a\vee\bar b}\vee \qquad \vdash c,\bar c}{\vdash (a\wedge b)\wedge c,\bar a\vee\bar b,\bar c}\wedge \qquad\qquad \dfrac{\dfrac{\dfrac{\vdash a,\bar a \quad \vdash b,\bar b}{\vdash a\wedge b,\bar a,\bar b}\wedge \qquad \vdash c,\bar c}{\vdash (a\wedge b)\wedge c,\bar a,\bar b,\bar c}\wedge}{\vdash (a\wedge b)\wedge c,\bar a\vee\bar b,\bar c}\vee$$

This kind of bureaucracy is present in all the usual deduction formalisms. One could imagine that there are simple ways to remove it: for instance quotienting proofs by some equivalence relation. However, this would not work, in particular, because logical rules are, in general, not linear.

The approach we develop in this paper makes use of the deep-inference methodology. Deep inference is a deduction framework (see [Gug07, BT01, Brü04]), where deduction rules apply arbitrarily deep inside formulae, contrary to traditional proof systems like natural deduction and sequent calculus, where deduction rules only deal with their outermost structure. The main reason to use deep inference is that it provides more freedom in designing proof systems, while maintaining the proof theoretic properties of interest, first and foremost cut elimination.

The simple principle of allowing inference to happen inside formulae leads to a natural change in the underlying structure of proofs, where rules are *unary*: one premiss and one conclusion. While proofs in usual deduction systems take the asymmetric form of a tree, deep inference allows to have a symmetric closure operation along the top-down axis. While sequent-calculus proofs cannot be dualised by flipping, this is always possible in deep inference, and it logically corresponds to dualities like the De Morgan one in classical logic.

A general methodology allows to design deep-inference deduction systems having more symmetries and finer structural properties than the sequent-calculus ones. For instance, cut and identity become really dual of each other, whereas they are only morally so in the sequent calculus, and all structural rules can be reduced to their atomic form, whereas contraction can not in the sequent calculus [Brü03].

All usual logics have deep-inference deduction systems enjoying cut elimination (see [Gug] for a complete overview). The standard proof system for propositional classical logic in deep inference is system SKS [BT01, Brü04]. The traditional methods of cut elimination of the sequent calculus can be adapted to a large extent to deep inference, despite having to cope with a higher generality [BT01, Brü04]. New methods are also achievable, based on weak computational traces of proofs called *atomic flows*. Atomic flows are directed acyclic graphs extracted from proofs that can be equipped with rewrite rules representing cut-elimination. Though being very simple (they trace only structural rules and forget logical rules), they are strong enough to faithfully represent and control cut-elimination procedures [GG08, Gun09], even a surprising quasipolynomial one [BGGP09].

So far, these developments have taken place inside a specific deep-inference formalism, dubbed the *calculus of structures*, where proofs are *sequential*, i.e., chains of formulae that are nothing else than terms in a term-rewriting chain generated by applications of unary rules of a given proof system. This very simple setting is not particularly intuitive and suffers from some forms of 'syntactic bureaucracy', like the one described before, where an irrelevant order of the rules is imposed by the formalism: this can be immediately appreciated by looking at the following two different proofs (that we take as logically equivalent, in some

unspecified logic):

$$\frac{\frac{A \wedge B}{C \wedge B}}{C \wedge D} \qquad \text{and} \qquad \frac{\frac{A \wedge B}{A \wedge D}}{C \wedge D} \quad ;$$

this is an obvious case of independent rewriting on two terms, but the sequential notion of proof does not allow for a canonical proof.

This paper shows that we can do better. The situation where the formalism imposes an irrelevant order of application of two rules to two independent subformulae is called bureaucracy of type A [Gug04a, Str09]. We define here a new formalism, called *open deduction*, that contains the calculus of structures as a special case and that provides a wider universe of proofs, where it is possible to normalise proofs into proofs where bureaucracy of type A is absent, using a simple procedure which is confluent and terminating. We call the proofs in this normal form *synchronal*. Referring to the example provided before, we have that open deduction allows the proof

$$\frac{A}{C} \wedge \frac{B}{D} \quad .$$

In fact, the definition of open deduction is based on a very simple, alternative but equivalent deep-inference notion to the term-rewriting one of the calculus of structures: proofs can be composed by connectives.

It should be emphasised that open deduction is a *logic-independent formalism*, which applies to all usual logics, thanks to its deep-inference foundation. Even if we are working at a high level of abstraction, we can still prove meaningful properties. In section 2, we exhibit a simple rewrite procedure that is confluent and terminating and that allows to transform any derivation into one in synchronal form, which is, moreover, of smaller size.

Of course, a natural question that pops up is: what happens to cut elimination? In particular, can we generalise the technique of atomic flows, that has been used in the particular case of the SKS system for classical propositional logic, to open deduction? We provide the basis of a positive answer in section 3. Thanks to deep inference, which allows us to represent logics with rules that are either *atomic* or *linear*, we define a general notion of atomic flows for open deduction. We show that the bureaucracy-elimination procedure of section 2, which transforms any open-deduction derivation into a synchronal one, has an important property: atomic flows are invariant under it.

In section 4, we restrict to system SKS for classical propositional logic and show that the rewrite rules of atomic flows, which are known to be sound with respect to sequential and synchronal derivations, are also sound with respect to open deductions in general. This means, in particular, that open-deduction cut elimination can be controlled by atomic flows the same way calculus-of-structures cut elimination is.

2. The Open Deduction Formalism

In this section, we present the open deduction formalism in a logic-independent way and illustrate it with the standard formalisation of propositional classical logic in deep inference. We define a canonical form of open deductions, called *synchronal*, which is free of the bureaucracy of type A described in the introduction. We prove that there is a

simple confluent and terminating rewriting procedure which transforms an arbitrary open deduction into a synchronal one.

Definition 2.1. We have the following mutually disjoint, countable sets:

- the set of *atoms* $\mathcal{A}$, whose elements are denoted by a, b, c and d (possibly with subscripts);
- for each $m < \omega$ and $n < \omega$, the set $\mathcal{R}_{m,n}$ of *logical relations* of *positive arity* m and *negative arity* n; we denote by $\mathcal{R}$, the set $\bigcup_{m,n\geq 0} \mathcal{R}_{m,n}$ of all logical relations; the elements of $\mathcal{R}$ are denoted by r (possibly with subscripts) and dedicated symbols for usual logical relations; the logical relations of positive and negative arity 0 are called *logical constants*.

Comment 1. The intended meaning of the positive and negative arities is that the following holds in the deduction system under consideration: if r is a logical relation of $\mathcal{R}_{m,n}$ and for each $i \leq n+m$, the formula B_i is deducible from the formula A_i then $r(B_1, \ldots, B_n, A_{n+1}, \ldots, A_{n+m})$ is deducible from $r(A_1, \ldots, A_n, B_{n+1}, \ldots, B_{n+m})$. The definition of derivation we will take in this paper ensures this property.
The connectives $\wedge$ and $\vee$ of classical logic are in $\mathcal{R}_{2,0}$, $\rightarrow$ is in $\mathcal{R}_{1,1}$ and $\neg$ is in $\mathcal{R}_{0,1}$.
It should be noted that there are also connectives of classical logic that do not satisfy the required property, for instance $\leftrightarrow$, but they can always be defined from connectives that do.

Definition 2.2. Let a set of logical relations $R = \bigcup_{m,n\geq 0} R_{m,n}$, where $R_{m,n} \subseteq \mathcal{R}_{m,n}$, be given.

1. The set $\mathcal{F}_{\mathcal{R}}$ of *formulae*, denoted by A, B, C and D (possibly with subscripts), is defined inductively by:
   (a) $\mathcal{A} \subseteq \mathcal{F}_R$;
   (b) $\mathcal{F}_R$ is closed by *logical relation composition*: if $r \in R_{m,n}$ and $A_1, \ldots, A_{m+n} \in \mathcal{F}_R$, then $r(A_1, \ldots, A_{m+n}) \in \mathcal{F}_R$.
2. The set $\mathcal{D}_R$ of *prederivations*, denoted by Φ and Ψ (possibly with subscripts), is defined inductively by :
   (a) $\mathcal{A} \subseteq \mathcal{D}_R$;
   (b) $\mathcal{D}_R$ is closed by *logical relation composition*: if $r \in R_{m,n}$ and $\Phi_1, \ldots, \Phi_{m+n} \in \mathcal{D}_R$, then $r(\Phi_1, \ldots, \Phi_{m+n}) \in \mathcal{D}_R$; and
   (c) $\mathcal{D}_R$ is closed by *inference composition*: if $\Phi_1, \Phi_2 \in \mathcal{D}_R$ then $\dfrac{\Phi_1}{\Phi_2} \in \mathcal{D}_R$.

   Inference composition is supposed to be associative.
3. The *premiss* and *conclusion* functions $\mathsf{pr}, \mathsf{cn} : \mathcal{D}_R \rightarrow \mathcal{F}_R$ are defined inductively as follows:
   (a) if $\Psi \in \mathcal{A}$, then $\mathsf{pr}\,\Psi = \mathsf{cn}\,\Psi = \Psi$;
   (b) if $r \in R$ and $\Psi = r(\Phi_1, \ldots, \Phi_m, \Phi'_1, \ldots, \Phi'_n)$, then
   $$\mathsf{pr}\,\Psi = r(\mathsf{pr}\,\Phi_1, \ldots, \mathsf{pr}\,\Phi_m, \mathsf{cn}\,\Phi'_1, \ldots, \mathsf{cn}\,\Phi'_n) \quad \text{and}$$
   $$\mathsf{cn}\,\Psi = r(\mathsf{cn}\,\Phi_1, \ldots, \mathsf{cn}\,\Phi_m, \mathsf{pr}\,\Phi'_1, \ldots, \mathsf{pr}\,\Phi'_n) \quad \text{; and}$$
   (c) if $\Psi = \dfrac{\Phi_1}{\Phi_2}$, then $\mathsf{pr}\,\Psi = \mathsf{pr}\,\Phi_1$ and $\mathsf{cn}\,\Psi = \mathsf{cn}\,\Phi_2$.
4. The sets of *positive contexts* and *negative contexts* are defined inductively as follows:
   (a) $\{\ \}$ is a positive context;

(b) if $r \in R_{m,n}$, $k \leq m$, A_k is a positive (resp. negative) context and for each $i \neq k$, A_i is a formula, then $r(A_1, \ldots, A_{m+n})$ is a positive (resp. negative) context;
(c) if $r \in R_{m,n}$, $m < k \leq m+n$, A_k is a positive (resp. negative) context and for each $i \neq k$, A_i is a formula, then $r(A_1, \ldots, A_{m+n})$ is a negative (resp. positive) context.

Contexts are denoted $K\{\ \}$. We use $K^+\{\ \}$ and $K^-\{\ \}$ when we need to specify the polarity of the context.

The *size* $|\Psi|$ of a prederivation (or formula or context) Ψ is the number of occurrences of atoms and logical relations in it.

Comment 2. Prederivations in open deduction have a natural planar representation where the inference composition is represented vertically and the logical relation composition is represented horizontally (see example 2.4)

Notation 1. For typographic convenience, inference composition of two prederivations Φ_1 and Φ_2 is also denoted by $\Phi_1|\Phi_2$. A prederivation Φ with $\mathsf{pr}\,\Phi = A$ and $\mathsf{cn}\,\Phi = B$ is denoted $\Phi : A \to B$ and represented in figures by

$$\begin{array}{c} A \\ {\scriptstyle\Phi}\Vert \\ B \end{array} .$$

Example 2.3. Classical propositional logic.

- The *logical relations* are:
  - *disjunction* $\vee$ and *conjunction* $\wedge$ which are in $\mathcal{R}_{2,0}$;
  - *negation* $\neg$ is which is in $\mathcal{R}_{0,1}$;
  - *logical constants*, f (false) and t (true), which are in $\mathcal{R}_{0,0}$.
- In the usual presentation of classical propositional logic, the SKS system, negation is not taken as a primitive connective, but defined by duality from its atomic case. The negation of an atom a is denoted $\bar{a}$. The disjunction and conjunction of two formulae A and B are denoted respectively $[A \vee B]$ and $(A \wedge B)$: the different brackets have the only purpose of improving legibility. We usually omit external brackets of formulae and sometimes we omit superfluous brackets under associativity. Example of formulae are $b \wedge [a \vee c]$ and $\neg [a \vee b] \wedge [a \vee c]$. An example of context $K\{\ \}$ is $b \wedge [\{\ \} \vee c]$; in this case $K\{a\}$ is $b \wedge [a \vee c]$, $K\{b\}$ is $b \wedge [b \vee c]$ and $K\{a \wedge d\}$ is $b \wedge [(a \wedge d) \vee c]$.

Example 2.4. The prederivation

$$\left(\begin{array}{c} a_1 \\ \hline a_2 \\ \hline a_3 \end{array} \wedge a_4 \right) \vee \neg \left[\begin{array}{c} \left(\dfrac{a_6}{a_5} \wedge a_7 \right) \vee a_9 \\ \hline a_8 \end{array} \right] .$$

has $(a_1 \wedge a_4) \vee \neg [a_8 \vee a_9]$ as premiss and $(a_3 \wedge a_4) \vee \neg [(a_6 \wedge a_7) \vee a_9]$ as conclusion.

Notation 2. If $K\{\ \}$ is a context and Φ a prederivation, we denote by $K\{\Phi\}$ the prederivation obtained by putting Φ in place of the hole in $K\{\ \}$. For example,

$$\text{if } K\{\ \} \text{ is } b \wedge [\{\ \} \vee c] \text{ and } \Phi \text{ is } \left(\dfrac{a_6}{a_5} \wedge a_7 \right), \text{ then } K\{\Phi\} \text{ is } b \wedge \left[\left(\dfrac{a_6}{a_5} \wedge a_7 \right) \vee c \right].$$

Definition 2.5. Given two prederivations $\Phi_1 : A \to B$ and $\Phi_2 : B \to C$, the *composition* of Φ_1 and Φ_2 , denoted $\Phi_1; \Phi_2 : A \to C$, is a prederivation defined inductively as follows:

- if $\Phi_1 \in \mathcal{A}$ then $\Phi_1; \Phi_2 \;=\; \Phi_2$,
- if $\Phi_1 \;=\; \dfrac{\Phi_1'}{\Phi_1''}$ then $\Phi_1; \Phi_2 \;=\; \dfrac{\Phi_1'}{\Phi_1''; \Phi_2}$
- if $\Phi_1 \;=\; r(\Phi^1_1, \ldots, \Phi^1_n, \Phi^1_{m+1}, \ldots \Phi^1_{m+n})$ with $r \in R_{n,m}$, then
  - if $\Phi_2 \;=\; \dfrac{\Phi_2'}{\Phi_2''}$ then $\Phi_1; \Phi_2 \;=\; \dfrac{\Phi_1; \Phi_2'}{\Phi_2''}$
  - if $\Phi_2 \;=\; r(\Phi^2_1, \ldots, \Phi^2_n, \Phi^2_{n+1}, \ldots \Phi^2_{n+m})$ then
  $$\Phi_1; \Phi_2 \;=\; r(\Phi^1_1; \Phi^2_1, \ldots, \Phi^1_n; \Phi^2_n, \Phi^2_{n+1}; \Phi^1_{n+1}, \ldots, \Phi^2_{n+m}; \Phi^1_{n+m})$$

Lemma 2.6. *The composition of two prederivations is well defined: the definition is compatible with associativity of inference composition. Moreover, though being given by an asymmetric double induction, the composition of two prederivations Φ_1 and Φ_2 is symmetric in the sense that:*

- $\Phi; a = a; \Phi = \Phi$ *(and more generally $\Phi; A = A; \Phi = \Phi$); and*
- $(\Phi_1|\Phi_2); \Psi = \Phi_1|(\Phi_2; \Psi)$ *and* $\Phi; (\Psi_1|\Psi_2) = (\Phi; \Psi_1)|\Psi_2$.

Notation 3. For typographic convenience, composition of two prederivations $\Phi_1 \colon A \to B$ and $\Phi_2 \colon B \to C$ is represented in figures by

$$\begin{array}{c}\Phi_1\\ \cdots\\ \Phi_2\end{array} \quad .$$

Lemma 2.7. $|\Phi_1; \Phi_2| = |\Phi_1| + |\Phi_2| - |\operatorname{cn} \Phi_1|$.

Lemma 2.8. *Composition of prederivations is associative.*

Proposition 2.9. *Given any two prederivations Φ and Ψ, we have:* $\Phi|\Psi = \Phi; (\operatorname{cn} \Phi|\operatorname{pr} \Psi); \Psi$.

Proof. By the previous lemmas, we have:
$\Phi|\Psi = (\Phi; \operatorname{cn} \Phi)|(\operatorname{pr} \Psi; \Psi) = \Phi; (\operatorname{cn} \Phi|(\operatorname{pr} \Psi; \Psi)) = \Phi; ((\operatorname{cn} \Phi|\operatorname{pr} \Psi); \Psi) = \Phi; (\operatorname{cn} \Phi|\operatorname{pr} \Psi); \Psi$. ■

Comment 3. The previous proposition states an important property: any prederivation can be 'decomposed' in such a way that inference composition only applies to formulae.

Definition 2.10. An *basic inference step* ρ is an inference composition $\dfrac{A}{B}$, where A and B are formulae called *premiss* and *conclusion*, respectively: it is denoted $\rho\,\dfrac{A}{B}$ or $A|_\rho B$. In concrete deduction systems, the set of basic inference steps is generated by a (finite) set of *inference rules* (with names for arbitrary atoms and formulae) from which the inference steps are instances.

If ρ is a basic inference step $\dfrac{A}{B}$, then the *flipped basic inference step* $\dfrac{B}{A}$ is denoted $\rho^\perp$, where $\cdot^\perp$ is an involution on the set of inference steps, i.e. $(\rho^\perp)^\perp = \rho$. If S is a set of basic inference steps, then $S^\perp$ is the set of basic inference steps $(\rho^\perp)$ for $\rho \in S$.

If $K\{\ \}$ is a positive (resp. negative) context and ρ is the basic inference step $\frac{A}{B}$, then we denote by $K\{\rho\}$ the *inference step* $\frac{K\{A\}}{K\{B\}}$ (resp. $\frac{K\{B\}}{K\{A\}}$).

The concept of derivation is obtained from the one of prederivation by restricting the inference composition to given inference steps.

Definition 2.11. Let a set of relations R and a set of basic inference steps S be given. The set $\mathcal{D}_{R,S}$ of *S-derivations* is defined inductively by

(1) $\mathcal{A} \subseteq \mathcal{D}_R$;
(2) if $r \in R_{m,n}$ and $\Phi_1, \dots, \Phi_{m+n} \in \mathcal{D}_R$, then $\Psi = r(\Phi_1, \dots, \Phi_{m+n}) \in \mathcal{D}_R$;
(3) if $\Phi_1, \Phi_2 \in \mathcal{D}_{R,S}$ then $\frac{\Phi_1}{\Phi_2} \in \mathcal{D}_{R,S}$ if there exists a context $K\{\ \}$ and a basic inference step ρ from S such that $\frac{\mathsf{cn}\,\Phi_1}{\mathsf{pr}\,\Phi_2}$ is the inference step $K\{\rho\}$.

Notation 4. An inference composition of Φ_1 and Φ_2 is denoted ${\scriptstyle K\{\rho\}}\frac{\Phi_1}{\Phi_2}$ or $\Phi_1|_{K\{\rho\}}\Phi_2$, where $K\{\rho\}$ is the inference step $\frac{\mathsf{cn}\,\Phi_1}{\mathsf{pr}\,\Phi_2}$. A S-derivation $\Phi : A \to B$ is denoted $\begin{matrix} A \\ {\scriptstyle \Phi}\|{\scriptstyle S} \\ B \end{matrix}$.
When it is clear from the context, we use the term *derivations* instead of *S-derivations* and omit S from the notation.

Definition 2.12. We define two canonical forms of *S-derivations*:

(1) the set of *sequential S-derivations* is the set of S-derivations where, in case (2) of Definition 2.11, $\Phi_1, \dots, \Phi_{n+m}$ are formulae.
(2) the set of *synchronal S-derivations* is the set of S-derivations where, in case (3) of Definition 2.11, $K\{\ \} = \{\ \}$, i.e. inference composition is restricted to basic inference steps.

Comment 4. Sequential derivations are the usual derivation of the calculus of structures. Synchronal derivations are derivations which are free of the type A bureaucracy described in the introduction (see example 2.15).

Lemma 2.13. *If Φ_1 and Φ_2 are S-derivations (resp. sequential S-derivations, synchronal S-derivations), then $\Phi_1; \Phi_2$ is a S-derivation (resp. sequential S-derivation, synchronal S-derivation).*

Example 2.14. The system SKS for classical propositional logic.
The set S of basic inference steps is the set of instances of the inference rules given below. The (usual) deep inference derivations of SKS are the sequential S-derivations.

Structural inference rules:

$$\begin{array}{ccc}
\mathsf{ai}\!\downarrow \dfrac{\mathsf{t}}{a \vee \bar{a}} & \mathsf{aw}\!\downarrow \dfrac{\mathsf{f}}{a} & \mathsf{ac}\!\downarrow \dfrac{a \vee a}{a} \\
\textit{identity (interaction)} & \textit{weakening} & \textit{contraction} \\
\mathsf{ai}\!\uparrow \dfrac{a \wedge \bar{a}}{\mathsf{f}} & \mathsf{aw}\!\uparrow \dfrac{a}{\mathsf{t}} & \mathsf{ac}\!\uparrow \dfrac{a}{a \wedge a} \\
\textit{cut (cointeraction)} & \textit{coweakening} & \textit{cocontraction}
\end{array},$$

Logical inference rules:

$$\begin{array}{cc}
\mathsf{s}\,\dfrac{A \wedge [B \vee C]}{(A \wedge B) \vee C} & \mathsf{m}\,\dfrac{(A \wedge B) \vee (C \wedge D)}{[A \vee C] \wedge [B \vee D]} \\
\textit{switch} & \textit{medial}
\end{array}.$$

In addition to these two rules, there are *equality* rules $=\dfrac{C}{D}$, for C and D in opposite sides in one of the following equations:

$$\begin{array}{cc}
A \vee B = B \vee A & A \vee \mathsf{f} = A \\
A \wedge B = B \wedge A & A \wedge \mathsf{t} = A \\
[A \vee B] \vee C = A \vee [B \vee C] & \mathsf{t} \vee \mathsf{t} = \mathsf{t} \\
(A \wedge B) \wedge C = A \wedge (B \wedge C) & \mathsf{f} \wedge \mathsf{f} = \mathsf{f}
\end{array}. \tag{2.1}$$

Comment 5. The SKS system shows an important property of deep inference formalism that we will use in the next section. The rules are of one of the two following kinds:

- *atomic rules*: only one atom name appears and no formula name appears.
- *linear rules*: each atom or formula name which appears in the premiss (resp. conclusion) appears exactly once in the conclusion (resp. premiss)

Example 2.15. We give here an example of a sequential derivation in SKS and its corresponding synchronal form.

$$\mathsf{m}\,\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{\mathsf{ac}\!\downarrow \dfrac{[(a \wedge b) \vee (a \wedge b)] \wedge [c \vee c]}{\mathsf{m}\,\dfrac{[(a \wedge b) \vee (a \wedge b)] \wedge c}{\mathsf{ac}\!\downarrow \dfrac{[a \vee a] \wedge [b \vee b] \wedge c}{\mathsf{ac}\!\downarrow \dfrac{a \wedge [b \vee b] \wedge c}{a \wedge b \wedge c}}}}} \qquad \mathsf{m}\,\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{\mathsf{m}\,\dfrac{(a \wedge b) \vee (a \wedge b)}{\dfrac{a \vee a}{a} \wedge \dfrac{b \vee b}{b}} \wedge \dfrac{c \vee c}{c}}$$

One can see on this example that the size of the synchronal derivation is smaller than the size of the sequential one.

Definition 2.16. (1) A *synchronisation redex* is an inference composition $K\{\rho\}\,\dfrac{\Phi_1}{\Phi_2}$ where $K\{\ \} \neq \{\ \}$. Note that thanks to proposition 2.9, a synchronisation redex can always be written $\Phi_1; (\mathsf{cn}\,\Phi_1|_{K\{\rho\}}\,\mathsf{pr}\,\Phi_2); \Phi_2$.

(2) The *contractum* of a synchronisation redex $\Phi_1; (\mathsf{cn}\,\Phi_1|_{K\{\rho\}}\,\mathsf{pr}\,\Phi_2); \Phi_2$ is $\Phi_1; K\{C|_\rho D\}; \Phi_2$, where C and D are the premiss and conclusion of the inference step ρ.

(3) The *synchronisation* reduction $\overset{\mathsf{sync}}{\to}$ on $\mathcal{D}_{\mathcal{R},S}$ is defined by: $\Phi \overset{\mathsf{sync}}{\to} \Psi$ iff Ψ is obtained from Φ by replacing a synchronisation redex by its contractum.

Lemma 2.17. *If Φ is a redex and Ψ its contractum, then $|\Psi| < |\Phi|$.*

Proof. Let $\Phi = \Phi_1; (\mathsf{cn}\,\Phi_1|_{K\{\rho\}}\,\mathsf{pr}\,\Phi_2); \Phi_2$ be a redex and $\Psi = \Phi_1; K\{C|_\rho D\}; \Phi_2$ its contractum. We have $|K\{C|_\rho D\}| = |K\{\ \}| + |C| + |D|$. By Lemma 2.7 we then have

$$|\Psi| = |\Phi_1| + |K\{\ \}| + |C| + |D| + |\Phi_2| - |K\{C\}| - |K\{D\}| = |\Phi_1| + |\Phi_2| - |K\{\ \}| < |\Phi|$$

∎

Theorem 2.18. *The synchronisation reduction is confluent and terminating. Moreover, each derivation reduces in a number of steps less than its size to its normal form which is a synchronal derivation.*

Proof. $\overset{\mathsf{sync}}{\to}$ is terminating because of Lemma 2.17. We show that $\overset{\mathsf{sync}}{\to}$ is locally confluent. Consider a derivation Ψ with two synchronisation redexes r_1 and r_2. Let Φ the smallest subderivation of Ψ which contains r_1 and r_2. There are two cases:
(1) $\Phi = r(\Phi_1, \ldots, \Phi_i, \ldots, \Phi_j, \ldots, \Phi_1)$ with r_1 in Φ_i, r_2 in Φ_j and $i \neq j$. Then the order of reduction of the two redexes obviously doesn't matter.
(2) $\Phi = \Phi_1|_\rho\Phi_2$. Then one of the following holds:

- One redex is in Φ_1 and the other in Φ_2. Then the order of reduction of the two redexes obviously doesn't matter.
- Φ is one of the redexes and the other one is in Φ_1 or Φ_2. Suppose for example that $\Phi = r_1$ and r_2 is in Φ_1. Thanks to proposition 2.9, we can write Φ as $\Phi_1; (\mathsf{cn}\,\Phi_1|_{K\{\rho\}}\,\mathsf{pr}\,\Phi_2); \Phi_2$. If the result of reducing r_2 in Φ_1 is Φ_1', then the result of reducing both redexes in any order is $\Phi_1'; K\{C|_\rho D\}; \Phi_2$.

The fact that a normal form is a synchronal derivation directly follows from definition 2.12.

∎

Comment 6. The reduction of a redex doesn't create any new redex. As a consequence, one can obtain the synchronal form by reducing all the redexes in parallel.

Example 2.19. We show here how the sequential derivation in Example 2.15 can be rewritten to the synchronal derivation in the same example by several applications of the $\overset{\mathsf{sync}}{\to}$ rewriting. The sequential derivation is written as a composition of inference steps, which is possible by 2.9, before the $\overset{\mathsf{sync}}{\to}$ rewriting is applied in parallel.

$$
\begin{array}{c}
\mathsf{m}\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{[(a \wedge b) \vee (a \wedge b)] \wedge [c \vee c]} \\
\cdots\cdots\cdots \\
\mathsf{ac}\!\downarrow\dfrac{[(a \wedge b) \vee (a \wedge b)] \wedge [c \vee c]}{[(a \wedge b) \vee (a \wedge b)] \wedge c} \\
\cdots\cdots\cdots \\
\mathsf{m}\dfrac{[(a \wedge b) \vee (a \wedge b)] \wedge c}{[a \vee a] \wedge [b \vee b] \wedge c} \\
\cdots\cdots\cdots \\
\mathsf{ac}\!\downarrow\dfrac{[a \vee a] \wedge [b \vee b] \wedge c}{a \wedge [b \vee b] \wedge c} \\
\cdots\cdots\cdots \\
\mathsf{ac}\!\downarrow\dfrac{a \wedge [b \vee b] \wedge c}{a \wedge b \wedge c}
\end{array}
\quad \overset{\mathsf{sync}\star}{\to} \quad
\begin{array}{c}
\mathsf{m}\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{[(a \wedge b) \vee (a \wedge b)] \wedge [c \vee c]} \\
\cdots\cdots\cdots \\
[(a \wedge b) \vee (a \wedge b)] \;\wedge\; \dfrac{c \vee c}{c} \\
\cdots\cdots\cdots \\
\mathsf{m}\dfrac{(a \wedge b) \vee (a \wedge b)}{[a \vee a] \wedge [b \vee b]} \;\wedge\; c \\
\cdots\cdots\cdots \\
\dfrac{a \vee a}{a} \;\wedge\; [b \vee b] \;\wedge\; c \\
\cdots\cdots\cdots \\
a \;\wedge\; \dfrac{b \vee b}{b} \;\wedge\; c
\end{array} .
$$

$$
\begin{array}{c}
= \\[1em]
\mathsf{m}\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{\mathsf{ac}\!\downarrow\dfrac{[(a \wedge b) \vee (a \wedge b)] \wedge [c \vee c]}{\mathsf{m}\dfrac{[(a \wedge b) \vee (a \wedge b)] \wedge c}{\mathsf{ac}\!\downarrow\dfrac{[a \vee a] \wedge [b \vee b] \wedge c}{\mathsf{ac}\!\downarrow\dfrac{a \wedge [b \vee b] \wedge c}{a \wedge b \wedge c}}}}}
\end{array}
\qquad
\begin{array}{c}
= \\[1em]
\mathsf{m}\dfrac{(a \wedge b \wedge c) \vee (a \wedge b \wedge c)}{\mathsf{m}\dfrac{(a \wedge b) \vee (a \wedge b)}{\dfrac{a \vee a}{a} \wedge \dfrac{b \vee b}{b}} \;\wedge\; \dfrac{c \vee c}{c}}
\end{array}
$$

Definition 2.20. Given a derivation Φ, we denote the normal form of Φ with respect to $\overset{\mathsf{sync}}{\to}$ by $\mathrm{sync}(\Phi)$.

Lemma 2.21. $\mathrm{sync}(\Phi;\Psi) = \mathrm{sync}(\Phi);\mathrm{sync}(\Psi)$.

Proof. We proceed by structural induction on Φ:

- the base case is trivial;
- $\mathrm{sync}((\Phi_1|_{K\{\rho\}}\Phi_2);\Psi)$
  $= \mathrm{sync}(\Phi_1|_{K\{\rho\}}(\Phi_2;\Psi))$
  $= \mathrm{sync}(\Phi_1);\mathrm{sync}(\mathsf{cn}\,\Phi_1|_{K\{\rho\}}\;\mathsf{pr}\,\Phi_2);\mathrm{sync}(\Phi_2;\Psi)$
  $= \mathrm{sync}(\Phi_1);\mathrm{sync}(\mathsf{cn}\,\Phi_1|_{K\{\rho\}}\;\mathsf{pr}\,\Phi_2);\mathrm{sync}(\Phi_2);\mathrm{sync}(\Psi)$
  $= \mathrm{sync}(\Phi);\mathrm{sync}(\Psi)$;
- when $\Phi = r(\Phi_1,\dots,\Phi_n)$, we proceed by structural induction on Ψ
  - if $\Psi = \Psi_1|\Psi_2$ we argue similarly to the previous case;
  - if $\Psi = r(\Psi_1,\dots,\Psi_n)$, we have that
    $\mathrm{sync}(\Phi;\Psi) = r(\mathrm{sync}(\Phi_1;\Psi_1),\dots,\mathrm{sync}(\Phi_n;\Psi_n)) = \mathrm{sync}(\Phi);\mathrm{sync}(\Psi)$.

∎

3. Atomic Flows

We now introduce a special kind of directed acyclic graphs, called *atomic flows*. Atomic flows associated with classical propositional derivations have been used to describe their normal forms, to define normalisation procedures on derivations and to prove properties

of these procedures. The atomic flow associated with a derivation represents the causal relationship between the creation and destruction of atoms in the derivation.

In this section we show that atomic flows can be associated with derivations in a logic-independent way, given certain very mild assumptions about the inference rules we use. We then show that atomic flows are invariants of the rewriting $\overset{\mathsf{sync}}{\to}$.

We first define atomic flows independently of derivations and deductive systems.

Definition 3.1. An *atomic flow* is a directed, acyclic graph $(V, E, \mathrm{up}, \mathrm{lo})$, such that

- V is a set of *vertices* and E a set of *edges*;
- $\mathrm{up}\colon E \to V \cup \{\top\}$ and $\mathrm{lo}\colon E \to V \cup \{\bot\}$ are, respectively, the *upper* and *lower* maps, where $\top, \bot \notin V$ and $\top \neq \bot$.

Atomic flows are denoted by ϕ and ψ (possibly with subscripts).
For every $\nu \in V \cup \{\top, \bot\}$, we define the set $L_\nu = \{\, \epsilon \mid \mathrm{up}(\epsilon) = \nu \,\}$ of *lower edges of* ν, the set $U_\nu = \{\, \epsilon \mid \mathrm{lo}(\epsilon) = \nu \,\}$ of *upper edges of* ν, and the set $E_\nu = L_\nu \cup U_\nu$ of *edges of* ν.
For an atomic flow ϕ, we call the set $U_\phi = L_\top$ (resp., $L_\phi = U_\bot$) the *upper* (resp., *lower*) *edges of* ϕ.

We can compose atomic flows similarly to how we compose derivations, and later we will see that the two notions work nicely together. We compose atomic flows by pairwise ‘identifying’ lower edges of one with upper edges of the other, according to a given one-to-one correspondence between the two.

Definition 3.2. Let $\phi_1 = (V_1, E_1, \mathrm{up}_1, \mathrm{lo}_1)$, $\phi_2 = (V_2, E_2, \mathrm{up}_2, \mathrm{lo}_2)$ be two atomic flows and f a bijection from a subset U'_{ϕ_2} of U_{ϕ_2} to a subset L'_{ϕ_1} of L_{ϕ_1}. The *composition* $\phi_1;_f \phi_2$ *of* ϕ_1 *and* ϕ_2 *with respect to* f is the flow $(V, E, \mathrm{up}, \mathrm{lo})$ defined as follows:

- the set of vertices V is the disjoint union of V_1 and V_2;
- the set of edges E is the disjoint union of E_1 and E_2, minus L_{ϕ_1};
- the up and lo maps of $\phi;_f \psi$ are inherited from the correspondind maps of ϕ_1 and ϕ_2 in the obvious way, except that, for every $\epsilon \in U'_{\phi_2}$, we have $\mathrm{up}(\epsilon) = \mathrm{up}_1(f(\epsilon))$.

Atomic flows are top-down symmetric, and in the same way that derivations are ‘flipped’ in a negative context, we might also want to ‘flip’ atomic flows. We now define the flipping operation.

Definition 3.3. The *flipping* operator $\cdot^\bot$ on atomic flows is defined as follows: if $\phi = (V, E, \mathrm{up}, \mathrm{lo})$ an atomic flow, then $\phi^\bot$ is the atomic flow $(V, E, \mathrm{up}^\bot, \mathrm{lo}^\bot)$ where, for every $\epsilon \in E$, if $\mathrm{up}(\epsilon) = \top$ (resp., $\mathrm{lo}(\epsilon) = \bot$), then $\mathrm{up}^\bot(\epsilon) = \bot$ (resp., $\mathrm{lo}^\bot(\epsilon) = \top$), otherwise $\mathrm{up}^\bot(\epsilon) = \mathrm{lo}(\epsilon)$ (resp., $\mathrm{lo}^\bot(\epsilon) = \mathrm{up}(\epsilon)$).

In deep inference most common logics can be expressed using only atomic and linear inference rules. In that case, we are able to separate the logical from the structural content of derivations, and atomic flows represent the structural content.

For the rest of this section we fix a set of logical relations $\mathcal{R}$ and a set of of basic inference steps $\mathcal{S} = \mathcal{S}_a \cup \mathcal{S}_l$, where $\mathcal{S}_a$ is a set of instances of atomic rules, $\mathcal{S}_l$ is a set of instances of linear rules.

Atomic flow associated to a derivation

Intuitively, every vertex of the atomic flow corresponds to an atomic inference rule or its converse, and its incident edges to the atom occurrences of this inference rule.

If Φ is a derivation, then $oc(\Phi)$ is the set of occurrences of atoms in Φ. We define in the following the *enriched atomic flow* $(\mathit{fl}(\Phi), f_\Phi^\top, f_\Phi^\perp)$ *of a derivation* $\Phi \in \mathcal{D}_{\mathcal{R},\mathcal{S}}$, where $\mathit{fl}(\Phi)$ is an atomic flow called the *atomic flow* of Φ and $f_\Phi^\top : oc(\mathsf{pr}(\Phi)) \to U_{\mathit{fl}(\Phi)}$ and $f_\Phi^\perp : oc(\mathsf{cn}(\Phi)) \to L_{\mathit{fl}(\Phi)}$ are bijections relating the upper (resp. lower) edges of the flow to the atom occurrences of the premiss (resp. conclusion) of the derivation.

(1) The enriched atomic flow $(\mathit{fl}(a), f_a^\top, f_a^\perp)$ of an atom a is defined as follows: $\mathit{fl}(a)$ is the flow consisting of no vertex and one edge, and $f_a^\top = f_a^\perp$ is the bijection mapping the atom to the edge.

(2) The enriched atomic flow $(\mathit{fl}(\Phi), f_\Phi^\top, f_\Phi^\perp)$ of a derivation $\Phi = r(\Phi_1, \dots, \Phi_m, \Phi'_1, \dots, \Phi'_n)$ with $r \in \mathcal{R}_{m,n}$ is defined as follows: $\mathit{fl}(\Phi)$ is the disjoint union of $\mathit{fl}(\Phi_1), \dots, \mathit{fl}(\Phi_m)$, $\mathit{fl}(\Phi'_1)^\perp, \dots, \mathit{fl}(\Phi'_n)^\perp$ and $f_\Phi^\top$ (resp., $f_\Phi^\perp$) is the disjoint union of $f_{\Phi_1}^\top, \dots, f_{\Phi_m}^\top, f_{\Phi'_1}^\perp, \dots, f_{\Phi'_n}^\perp$ (resp., $f_{\Phi_1}^\perp, \dots, f_{\Phi_m}^\perp, f_{\Phi'_1}^\top, \dots, f_{\Phi'_n}^\top$)

(3) The enriched atomic flow $(\mathit{fl}(\rho), f_\Phi^\top, f_\Phi^\perp)$ of a basic inference step ρ is defined as follows:
   - If ρ is an instance $C|D$ of a linear rule, then $\mathit{fl}(\rho) = \mathit{fl}(C)$, $f_\rho^\top = f_C^\top$ and $f_\rho^\perp = g \circ f_C^\perp$, where g is the bijection between the occurrences of atoms in D and the corresponding occurrences in C, which exits thanks to the linearity of the rule.
   - If ρ is an instance $C|D$ of an atomic rule, then $\mathit{fl}(\rho) = (\{v\}, U_{\mathit{fl}(C)} + L_{\mathit{fl}(D)}, \mathrm{up}, \mathrm{lo})$, where up and lo are defined as follows:
     - $\mathrm{up}(e)$ is v, if $e \in U_{\mathit{fl}(C)}$ and $\top$, otherwise.
     - $\mathrm{lo}(e)$ is v, if $e \in L_{\mathit{fl}(D)}$ and $\perp$, otherwise.

   The flow ϕ is enriched by taking $f_\rho^\top = f_C^\top$ and $f_\rho^\perp = f_D^\perp$.

(4) The enriched atomic flow $(\mathit{fl}(\Phi), f_\Phi^\top, f_\Phi^\perp)$ of a derivation $\Phi = \Phi_1|_{K\{\rho\}}\Phi_2$ is defined as follows. Suppose that ρ is $C|D$. We write Φ as $\Phi_1; K\{C\}|K\{D\}; \Phi_2$ if $K\{\ \}$ is positive and Φ as $\Phi_1; K\{D\}|K\{C\}; \Phi_2$ if $K\{\ \}$ is negative. We then obtain the flow of Φ by composing the flows of the compound derivations:
   - $\mathit{fl}(\Phi) = \mathit{fl}(\Phi_1);_g \mathit{fl}(K\{C|D\});_h \mathit{fl}(\Phi_2)$, where $g = f_{\Phi_1}^\perp \circ (f_{\mathit{fl}(K\{C|D\})}^\top)^{-1}$ and $h = f_{\mathit{fl}(K\{C|D\})}^\perp \circ (f_{\Phi_2}^\top)^{-1}$;
   - $f_\Phi^\top = f_{\Phi_1}^\top$
   - $f_\Phi^\perp = f_{\Phi_2}^\perp$

Examples of atomic flows associated to derivations

We consider atomic flows for classical propositional derivations in SKS [GG08].

We first give the atomic flows associated to the basic inference steps associated to structural rules of SKS, from which the flows of all the SKS derivations are build. Vertices corresponding to the structural rules are represented by three different symbols (—, ▽ and △) and their incident edges (represented by vertical lines) correspond to the atom occurrences of the rules. The labels of the occurrences of atoms are also indicated on the edges they correspond to, in order to ease the reading.

$$\mathsf{ai}\!\downarrow \frac{\mathsf{t}}{a^{\mathbf{1}} \vee \bar{a}^{\mathbf{2}}} \quad \to \quad \qquad \mathsf{aw}\!\downarrow \frac{\mathsf{f}}{a^{\mathbf{1}}} \quad \to \quad \qquad \mathsf{ac}\!\downarrow \frac{a^{\mathbf{1}} \vee a^{\mathbf{2}}}{a^{\mathbf{3}}} \quad \to$$

$$\mathsf{ai}\!\uparrow \frac{a^{\mathbf{1}} \wedge \bar{a}^{\mathbf{2}}}{\mathsf{f}} \quad \to \quad \qquad \mathsf{aw}\!\uparrow \frac{a^{\mathbf{1}}}{\mathsf{t}} \quad \to \quad \qquad \mathsf{ac}\!\uparrow \frac{a^{\mathbf{3}}}{a^{\mathbf{1}} \wedge a^{\mathbf{2}}} \quad \to$$

Here is now the example of a flow associated to a general inference step, which consist of an application of a basic inference step in a context. The context creates edges not related to vertices.

$$\mathsf{ac}\!\downarrow \frac{a^{\mathbf{1}} \wedge \left[b^{\mathbf{2}} \vee \left[a^{\mathbf{3}} \vee a^{\mathbf{4}}\right]\right]}{a^{\mathbf{1}} \wedge \left[b^{\mathbf{2}} \vee a^{\mathbf{5}}\right]} \quad \to \quad .$$

We finally give the example of a flow associated to to a sequential derivation.

$$\begin{array}{l} \mathsf{ai}\!\downarrow \dfrac{\left(a^{\mathbf{1}} \wedge \left[\bar{a}^{\mathbf{3}} \vee \mathsf{t}\right]\right) \wedge \bar{a}^{\mathbf{8}}}{\left(a^{\mathbf{1}} \wedge \left[\bar{a}^{\mathbf{3}} \vee \left[\bar{a}^{\mathbf{4}} \vee a^{\mathbf{5}}\right]\right]\right) \wedge \bar{a}^{\mathbf{8}}} \\ = \dfrac{}{\left(a^{\mathbf{1}} \wedge \left[\left[\bar{a}^{\mathbf{3}} \vee \bar{a}^{\mathbf{4}}\right] \vee a^{\mathbf{5}}\right]\right) \wedge \bar{a}^{\mathbf{8}}} \\ \mathsf{s} \dfrac{}{\left[\left(a^{\mathbf{1}} \wedge \left[\bar{a}^{\mathbf{3}} \vee \bar{a}^{\mathbf{4}}\right]\right) \vee a^{\mathbf{5}}\right] \wedge \bar{a}^{\mathbf{8}}} \\ \mathsf{ac}\!\downarrow \dfrac{}{\left[\left(a^{\mathbf{1}} \wedge \bar{a}^{\mathbf{2}}\right) \vee a^{\mathbf{5}}\right] \wedge \bar{a}^{\mathbf{8}}} \\ \mathsf{ai}\!\uparrow \dfrac{}{\left[\mathsf{f} \vee a^{\mathbf{5}}\right] \wedge \bar{a}^{\mathbf{8}}} \\ = \dfrac{}{a^{\mathbf{5}} \wedge \bar{a}^{\mathbf{8}}} \\ \mathsf{ac}\!\uparrow \dfrac{}{\left(a^{\mathbf{6}} \wedge a^{\mathbf{7}}\right) \wedge \bar{a}^{\mathbf{8}}} \\ = \dfrac{}{a^{\mathbf{6}} \wedge \left(a^{\mathbf{7}} \wedge \bar{a}^{\mathbf{8}}\right)} \\ \mathsf{ai}\!\uparrow \dfrac{}{a^{\mathbf{6}} \wedge \mathsf{f}} \end{array} \quad \to$$

By design the composition of flows and the composition of derivations work together as expected:

Lemma 3.4. *For any atomic flows $\Phi\colon A \to B$ and $\Psi\colon B \to C$, we have that $\mathit{fl}(\Phi;\Psi) = \mathit{fl}(\Phi);_{f_\Phi^\perp \circ (f_\Psi^\top)^{-1}} \mathit{fl}(\Psi)$.*

Atomics flows have been defined for sequential derivations in [GG08] and for synchronal derivations in [Gun09], in the case of classical propositionnal derivations in SKS. We now show that the two notions coincide in general, and that atomic flows are in fact invariants of the $\overset{\mathsf{sync}}{\to}$ rewriting.

Theorem 3.5. *If $\Phi \overset{\mathsf{sync}}{\to} \Psi$, then $\mathit{fl}(\Phi) = \mathit{fl}(\Psi)$.*

Sketch of proof. We first remark that $\mathit{fl}(K\{A'\}|_{K\{\rho\}}K\{B'\}) = \mathit{fl}(K\{A'|_\rho B'\})$. It then follows by the previous lemma that the flow of a synchronisation redex is the same as the flow of its contractum. The result is obtained by checking that the flow of a derivation is the same as the flow of the result of replacing a synchronisation redex by its contractum. ∎

Example 3.6. The sequential derivation on the left reduces to the synchronous derivation on the right, and they both have the atomic flow in the middle, where the correspondence between atom occurrences and edges is indicated by colours:

$$
\begin{array}{rc}
 & (a \wedge [\bar{a} \vee \mathsf{t}]) \wedge \bar{a} \\
\mathsf{ai}\!\downarrow & \overline{(a \wedge [\bar{a} \vee [\bar{a} \vee a]]) \wedge \bar{a}} \\
= & \overline{(a \wedge [[\bar{a} \vee \bar{a}] \vee a]) \wedge \bar{a}} \\
\mathsf{s} & \overline{[(a \wedge [\bar{a} \vee \bar{a}]) \vee a] \wedge \bar{a}} \\
\mathsf{ac}\!\downarrow & \overline{[(a \wedge \bar{a}) \vee a] \wedge \bar{a}} \\
\mathsf{ai}\!\uparrow & \overline{[\mathsf{f} \vee a] \wedge \bar{a}} \\
= & \overline{a \wedge \bar{a}} \\
\mathsf{ac}\!\uparrow & \overline{(a \wedge a) \wedge \bar{a}} \\
= & \overline{a \wedge (a \wedge \bar{a})} \\
\mathsf{ai}\!\uparrow & \overline{a \wedge \mathsf{f}}
\end{array}
\qquad
= \frac{\left(\mathsf{s}\,\dfrac{a \wedge \left[\bar{a} \vee \dfrac{\mathsf{t}}{\bar{a} \vee a} \right]}{\dfrac{a \wedge \dfrac{\bar{a} \vee \bar{a}}{\bar{a}}}{\mathsf{f}} \vee \dfrac{a}{a \wedge a}} \wedge \bar{a} \right)}{a \wedge \dfrac{a \wedge \bar{a}}{\mathsf{f}}} \; .
$$

4. Atomic Flow Rewriting

We now give an example of the use of atomic flows with respect to the system SKS for propositional classical logic. We define reductions on flows and then we show how they can be lifted to derivations. This has been done for sequential derivations in [GG08] and for synchronal derivations in [Gun09]: here we show that the reductions on synchronal derivations correspond exactly to reductions on sequential derivations modulo $\overset{\mathsf{sync}}{\to}$.

Definition 4.1. We define graphical expressions of the kind $r\colon \phi' \to \psi'$, where r is a name and ϕ' and ψ' are flows:

w↓-i↑: $\to$ i↓-w↑: $\to$

w↓-c↑: $\to$ c↓-w↑: $\to$

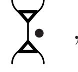
w↓-w↑: $\to$ c↓-c↑: $\to$.

c↓-i↑: $\to$ i↓-c↑: $\to$

Definition 4.2. For every expression $r\colon \phi' \to \psi'$ from Definition 4.1, the reduction $\to_r$ is defined, such that $\phi \to_r \psi$ if and only if ϕ' is a subflow in ϕ and we obtain ψ by replacing ϕ' with ψ' in ϕ, while respecting the correspondence of edges.

Theorem 4.3. *For each* $r \in \{\mathsf{w}\!\downarrow\text{-}\mathsf{i}\!\uparrow, \mathsf{i}\!\downarrow\text{-}\mathsf{w}\!\uparrow, \mathsf{w}\!\downarrow\text{-}\mathsf{c}\!\uparrow, \mathsf{c}\!\downarrow\text{-}\mathsf{w}\!\uparrow, \mathsf{w}\!\downarrow\text{-}\mathsf{w}\!\uparrow, \mathsf{c}\!\downarrow\text{-}\mathsf{c}\!\uparrow, \mathsf{c}\!\downarrow\text{-}\mathsf{i}\!\uparrow, \mathsf{i}\!\downarrow\text{-}\mathsf{c}\!\uparrow\}$ *and every* SKS*-derivation (resp., synchronal or sequential* SKS*-derivation)* $\Phi\colon A \to B$ *and every atomic flow* ψ*, such that* $\mathit{fl}(\Phi) \to_r \psi$*; there exists an* SKS*-derivation (resp., synchronal or sequential* SKS*-derivation)* $\Psi\colon A \to B$ *with flow* ψ*.*

Proof. We consider the case for c↓-c↑, the other cases can be proven similarly. Assuming *fl*(Φ) contains



let every atom occurrence a in Φ that is mapped to the edge labelled with • be labelled $a^\bullet$. By Proposition 2.9 Φ must contain the two subderivations $\Phi' = \Phi'_1; \left(\mathsf{ac}{\downarrow}\,\dfrac{K_1\,[a \vee a]}{K_1\{a^\bullet\}}\right); \Phi'_2$ and $\Phi'' = \Phi''_1; \left(\mathsf{ac}{\uparrow}\,\dfrac{K_2\{a^\bullet\}}{K_2\,(a \wedge a)}\right); \Phi''_2$.

If Φ is synchronal, we have that $K_1\{\ \} = K_2\{\ \} = \{\ \}$ and we define

$$\hat{\Psi}' \quad = \quad \mathsf{m}\,\dfrac{\left[\dfrac{a}{a \wedge a} \vee \dfrac{a}{a \wedge a}\right]}{[a \vee a] \wedge [a \vee a]} \quad \text{and} \quad \hat{\Psi}'' \quad = \quad \left(\dfrac{a \vee a}{a} \wedge \dfrac{a \vee a}{a}\right) \quad .$$

Otherwise, we define

$$\hat{\Psi}' \quad = \quad \mathsf{m}\,\dfrac{\mathsf{ac}{\uparrow}\,\dfrac{\mathsf{ac}{\uparrow}\,\dfrac{K_1\,[a \vee a]}{K_1\,[(a \wedge a) \vee a]}}{K_1\,[(a \wedge a) \vee (a \wedge a)]}}{K_1\,([a \vee a] \wedge [a \vee a])} \quad \text{and} \quad \hat{\Psi}'' \quad = \quad \mathsf{ac}{\downarrow}\,\dfrac{\mathsf{ac}{\downarrow}\,\dfrac{K_2\,([a \vee a] \wedge [a \vee a])}{K_2\,([a \vee a] \wedge a)}}{K_2\,(a \wedge a)} \quad .$$

Finally, we define

$$\Psi' = \Phi'_1; \hat{\Psi}'; \Phi'_2\{a^\bullet / ([a \vee a] \wedge [a \vee a])\} \quad \text{and} \quad \Psi'' = \Phi''_1\{a^\bullet / ([a \vee a] \wedge [a \vee a])\}; \hat{\Psi}''; \Phi''_2 \quad .$$

This allows us to obtain the derivation $\Psi\colon A \to B$ with the required atomic flow from Φ, by simultaneously applying the substitution $\{a^\bullet / ([a \vee a] \wedge [a \vee a])\}$, replacing Φ' with Ψ', and replacing Φ'' with Ψ''. ■

Definition 4.4. Given $r \in$ {w↓-i↑, i↓-w↑, w↓-c↑, c↓-w↑, w↓-w↑, c↓-c↑, c↓-i↑, i↓-c↑}, an SKS-derivation Φ, a flow ψ, such that *fl*(Φ) $\to_r \psi$, and the SKS-derivation Ψ constructed in the proof of Theorem 4.3, we write $\Phi \to_r \Psi$.

Theorem 4.5. *Given* SKS*-derivations* Φ_1 *and* Φ_2*, such that* $\mathrm{sync}(\Phi_1) = \mathrm{sync}(\Phi_2)$*, then if* $\Phi_1 \to_r \Psi_1$ *and* $\Phi_2 \to_r \Psi_1$ *for some* $r \in$ {w↓-i↑, i↓-w↑, w↓-c↑, c↓-w↑, w↓-w↑, c↓-c↑, c↓-i↑, i↓-c↑}*, we have* $\mathrm{sync}(\Psi_1) = \mathrm{sync}(\Psi_2)$*.*

Sketch of proof. We sketch the proof for $r =$ c↓-c↑, the other cases can be proved similarly: The result follows by Lemma 2.21 and the fact that

$$\dfrac{\dfrac{\dfrac{K_1\,[a \vee a]}{K_1\,[(a \wedge a) \vee a]}}{K_1\,[(a \wedge a) \vee (a \wedge a)]}}{K_1\,([a \vee a] \wedge [a \vee a])} \quad \overset{\mathsf{sync}\star}{\to} \quad K_1\left\{\dfrac{\dfrac{a}{a \wedge a} \vee \dfrac{a}{a \wedge a}}{[a \vee a] \wedge [a \vee a]}\right\} \quad \text{and}$$

$$\dfrac{\dfrac{K_2\,([a \vee a] \wedge [a \vee a])}{K_2\,([a \vee a] \wedge a)}}{K_2\,(a \wedge a)} \quad \overset{\mathsf{sync}\star}{\to} \quad K_2\left(\dfrac{a \vee a}{a} \wedge \dfrac{a \vee a}{a}\right) \quad .$$

■

5. Conclusions

We have seen, in this paper, that we can reduce the syntactic bureaucracy of proof systems using a logic-independent formalism, without sacrificing the usual proof-theoretic tools and properties like cut-elimination and complexity. We eliminate 'type A' bureaucracy, i.e., the irrelevant order of application of two rules to two independent subformulae [Gug04a].

The formalism here introduced, called *open deduction*, generalises the calculus of structures, which in turn generalises the sequent calculus and natural deduction. This generality does not claim a price on the naturalness of the formalism: it simply extends to proofs the structure of formulae, that is only partly used in the sequent calculus.

However, the necessary technology to deal with cut elimination is not trivial, and only now we are in a position to deal with it, relying on recent developments of the calculus of structures. This paper shows that if we can normalise a proof in the calculus of structures, then we can eliminate its bureaucracy in open deduction. Moreover, since the atomic flow of proofs is invariant under bureaucracy elimination, we can start developing atomic-flow-based cut elimination directly in open deduction.

In a forthcoming paper, we will define an even more general formalism, containing open deduction and eliminating a further type of bureaucracy, namely the irrelevant order of application of two rules to two nested subformulae. This is dubbed 'type B' bureaucracy in [Gug04b]. The reason to separate, at this stage, open deduction and its further generalisation, is that addressing type B bureaucracy requires a level of abstraction that would be best supported by further developments of atomic flows, to which our efforts are dedicated.

References

[BGGP09] Paola Bruscoli, Alessio Guglielmi, Tom Gundersen, and Michel Parigot. A quasipolynomial cut-elimination procedure in deep inference via atomic flows and threshold formulae. Submitted. `http://cs.bath.ac.uk/ag/p/QPNDI.pdf`, 2009.

[Brü03] 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`.

[Brü04] Kai Brünnler. *Deep Inference and Symmetry in Classical Proofs*. Logos Verlag, Berlin, 2004. `http://www.iam.unibe.ch/~kai/Papers/phd.pdf`.

[BT01] 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`.

[GG08] 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`.

[Gug] Alessio Guglielmi. Deep inference. Web site at `http://alessio.guglielmi.name/res/cos`.

[Gug04a] Alessio Guglielmi. Formalism A. `http://cs.bath.ac.uk/ag/p/AG11.pdf`, 2004.

[Gug04b] Alessio Guglielmi. Formalism B. `http://cs.bath.ac.uk/ag/p/AG13.pdf`, 2004.

[Gug07] Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1):1–64, 2007. `http://cs.bath.ac.uk/ag/p/SystIntStr.pdf`.

[Gun09] Tom Gundersen. *A General View of Normalisation Through Atomic Flows*. PhD thesis, University of Bath, 2009.

[Str09] Lutz Straßburger. From deep inference to proof nets via cut elimination. *Journal of Logic and Computation*, 2009. In press. `http://www.lix.polytechnique.fr/~lutz/papers/deepnet.pdf`.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 151-160
http://rewriting.loria.fr/rta/

A REWRITING LOGIC SEMANTICS APPROACH TO MODULAR PROGRAM ANALYSIS

MARK HILLS [1] AND GRIGORE ROŞU [2]

[1] Centrum Wiskunde & Informatica
Science Park 123, 1098 XG Amsterdam, The Netherlands
E-mail address: Mark.Hills@cwi.nl
URL: http://www.cwi.nl

[2] Department of Computer Science, University of Illinois at Urbana-Champaign
201 N. Goodwin Av., Urbana, IL 61801, USA
E-mail address: grosu@cs.uiuc.edu
URL: http://fsl.cs.uiuc.edu

ABSTRACT. The K framework, based on rewriting logic semantics, provides a powerful logic for defining the semantics of programming languages. While most work in this area has focused on defining an evaluation semantics for a language, it is also possible to define an abstract semantics that can be used for program analysis. Using the SILF language (Hills, Serbanuta and Rosu, 2007), this paper describes one technique for defining such a semantics: policy frameworks. In policy frameworks, an analysis-generic, modular framework is first defined for a language. Individual analyses, called policies, are then defined as extensions of this framework, with each policy defining analysis-specific semantic rules and an annotation language which, in combination with support in the language front-end, allows users to annotate program types and functions with information used during program analysis. Standard term rewriting techniques are used to analyze programs by evaluating them in the policy semantics.

1. Introduction

Programs compute by manipulating different kinds of *explicit* data made available by the language, like integers, objects, lists, functions, or strings. Some of this data may also have *implicit* properties, important to the correctness of the program but impossible to represent directly in the language. For example, many languages have no way to indicate that a variable has been (or must already be) explicitly initialized, or that a reference or pointer never contains null. Some languages also leave the types of values and variables implicit, providing no syntax to indicate that certain types are expected at given points in the

1998 ACM Subject Classification: F.3.2 [Semantics of Programming Languages]: Program Analysis.
Key words and phrases: K, rewriting logic semantics, program analysis.
Supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, by the Microsoft/Intel funded Universal Parallel Computing Research Center at UIUC, and by several Microsoft gifts.

© M. Hills and G. Roşu
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.151

program. Domain-specific examples of implicit properties are also common. One compelling example, commonly used in scientific computing applications, is units of measurement [nis], where program values and variables are assumed to have specific units (meters, kilograms, seconds, etc) at specific points in the program or along specific execution paths.

These implicit properties of program data give rise to implicit *policies*, or rules about how this information can be manipulated. For instance, one may require that variables be initialized on all paths before being read, or that only non-null pointers can be assigned to other non-null pointers. Languages with implicit types generally still place type restrictions on operations such as arithmetic, where only values representing numbers can be used. Programs that use units of measurement must adhere to a number of rules, such as requiring two operands in an addition or comparison operation to have the same unit, or treating the result of a multiplication operation as having a unit equal to the product of the units of the operands (e.g., given `meter` and `second`, the resulting unit would be `meter second`).

Because these properties are hard to check by hand, a number of techniques have been developed to allow implicit properties to be either inferred or stated explicitly in a program. In this paper we focus on the use of *annotations*, either given by "decorating" program constructs with type-like information (type annotations) or by including additional information in special language constructs or inside comments (code annotations). Many systems that use annotations are designed with specific analysis domains in mind; those that are more general often support either type or code annotations, but not both, or provide limited capabilities to adapt to new domains. In this paper we present a solution designed to overcome these limitations: *policy frameworks*. Policy frameworks support the use of type and code annotations through an augmented language front-end, with each analysis *policy* defining its own annotation language, specialized to the domain under analysis. Program analysis is then based on program evaluation in an abstract rewriting logic semantics of the programming and annotation languages. Principles developed during work on the rewriting logic semantics project are used to ensure that the semantics is modular, allowing a large core of the framework to be reused across policies.

The remainder of this paper is organized as follows. As background, Section 2 provides an introduction to rewriting logic semantics and K, a rewrite-based formalism for language semantics. Section 3 then presents a policy framework for the SILF programming language along with two policies. SILF has been chosen because it is complex enough to show many common features of programming languages, but simple enough to allow policy frameworks to be understood in isolation from the language, something that is more difficult for languages such as C. Finally, Section 4 discusses related work, while Section 5 concludes.

2. Rewriting Logic Semantics and K

Equational logic has long been seen as a viable formalism for defining the semantics of sequential programming languages [Gog77, Ber89, Gog96]. Rewriting logic extends this by providing a formalism for defining the semantics of nondeterministic and concurrent languages, leading to an area of research known as rewriting logic semantics [Mes04, Mes07]. One specific style of rewriting logic semantics is computation-based rewriting logic semantics [Mes07], hereafter referred to as RLS.

RLS defines the semantics of a programming language as a rewrite theory. Terms formed over the signature of the theory are used to represent the program *configuration*, made up of the current program and auxiliary entities such as environments, stores, etc.

Rules and equations are then used to transition between configurations, with equations used to define sequential language features and rules used to define nondeterministic and concurrent language features. The configuration is defined as a nested multiset: individual parts of the configuration (*configuration items*) can be repeated and can also be nested inside other items, allowing the flexibility to represent language features such as multiple threads (repetition), each with local state (nesting). Multiset matching is used so that configuration items do not need to be named in a specific order in equations and rules; matching also allows unused parts of the configuration around the matched subterm to be elided, allowing equations and rules to remain unchanged even when the surrounding configuration changes.

One often-used configuration item, `k`, holds the current computation. Computations in `k` are lists; each item in the list is referred to as a computation item, each of which represents an individual task or piece of information. The head of the list can be seen as the "next" task, with the tail containing tasks that will be computed later. Instead of using "`,`" as the list separator, an arrow, written in text as `->` and mathematically as $\curvearrowright$, is used, hopefully providing some added intuition: do this (ci_1), then that (ci_2), then that (ci_3), etc, until finished (no items are left):

$$ci_1 \curvearrowright ci_2 \curvearrowright ci_3 \curvearrowright \ldots \curvearrowright ci_n$$

The equations and rules used to define the semantics often break up computations into smaller pieces, which are then put at the head of the computation to indicate that they need to be computed first before the overall computation can continue. Since computations in RLS are first-order terms, they can also be manipulated as a whole, such as by saving the current computation for evaluation later (such as for call/cc or coroutines).

```
eq stmt(E ;) = exp(E) -> discard .
eq exp(X) = lookup(X) .
eq stmt(X <- E ;) = exp(E) -> assignTo(X) .
eq k(val(V) -> assignTo(X) -> K) env([X,L] Env) mem([L,V'] Mem) =
   k(                          K) env([X,L] Env) mem([L,V ] Mem) .
```

Figure 1: RLS Semantics with Equations

Figure 1 shows several examples of equations. The first provides the semantics for the statement `E ;`, saying that this is defined as the result of evaluating the expression `E` and then discarding the result. Note that `exp(E)` is placed "on top of" `discard` in the computation, meaning that it will be evaluated first, with the expectation that it will produce a value. The second equation provides a semantics for names used as expressions: `X` is looked up to retrieve its current value. The third equation provides the semantics for assignment: `E` is evaluated, and the resulting value is assigned to `X` using `assignTo`. Finally, the fourth equation defines the semantics of `assignTo`. `Env` and `Mem` are both multisets of pairs, defined equationally as maps – `Env` from names to locations, `Mem` from locations to values. The equation states that the result of assigning value `V` to name `X` is a configuration where the value `V'` held at location `L` in `Mem` – location `L` is assigned to `X` by `Env` – is replaced with the new value `V`.

K. K [Ros07], based on RLS, provides additional notation for defining the semantics of a language. K configurations are defined identically to those in RLS, with each configuration item called a K *cell*. The cells are given in K rules using an XML-like notation, with an opening cell "tag", like $\langle k \rangle$ and a closing tag like $\langle /k \rangle$. Rules in K are defined similarly to rules and equations in RLS, but with a number of notational conveniences. Figure 2 shows the step-by-step results of a number of individual transformation steps to convert the last RLS equation in Figure 1 (augmented with support for threads) into a K rule.

$$\langle t\rangle\ \langle k\rangle\ X\ \leftarrow\ V \curvearrowright K\ \langle/k\rangle\langle env\rangle\ (X,L)\ Env\ \langle/env\rangle\ TS\ \langle/t\rangle \quad \langle mem\rangle\ (L,V')\ Mem\ \langle/mem\rangle \rightarrow \\ \langle t\rangle\ \langle k\rangle\ K\ \langle/k\rangle\langle env\rangle\ (X,L)\ Env\ \langle/env\rangle\ TS\ \langle/t\rangle \quad \langle mem\rangle\ (L,V)\ Mem\ \langle/mem\rangle \tag{2.1}$$

$$\langle k\rangle\ X\ \leftarrow\ V \curvearrowright K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,V')\ Mem\ \langle/mem\rangle \rightarrow \\ \langle k\rangle\ K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,V)\ Mem\ \langle/mem\rangle \tag{2.2}$$

$$\langle k\rangle\ X\ \leftarrow\ V \curvearrowright K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,_)\ Mem\ \langle/mem\rangle \rightarrow \\ \langle k\rangle\ K\ \langle/k\rangle \quad \langle env\rangle\ (X,L)\ Env\ \langle/env\rangle \quad \langle mem\rangle\ (L,V)\ Mem\ \langle/mem\rangle \tag{2.3}$$

$$\langle k\rangle\ X\ \leftarrow\ V\ \ ...\langle/k\rangle \quad \langle env\rangle...\ \ (X,L)\ \ ...\langle/env\rangle \quad \langle mem\rangle...\ \ (L,_)\ \ ...\langle/mem\rangle \rightarrow \\ \langle k\rangle\ \cdot\ \ ...\langle/k\rangle \quad \langle env\rangle...\ \ (X,L)\ \ ...\langle/env\rangle \quad \langle mem\rangle...\ \ (L,V)\ \ ...\langle/mem\rangle \tag{2.4}$$

$$\langle k\rangle \underset{\cdot}{\underline{X\ \leftarrow\ V}}\ ...\langle/k\rangle\ \langle env\rangle...\ \ (X,L)\ \ ...\langle/env\rangle\ \langle mem\rangle...\ \ (L,\underset{V}{\underline{_}})\ \ ...\langle/mem\rangle \tag{2.5}$$

Figure 2: Converting RLS to K

The first step of the transformation is shown in Rule 2.1. The main change in this first step is the use of the XML-like cell notation and the replacement of `val(V) -> assignTo(X)` with $X \leftarrow V$. The second step, shown in Rule 2.2, is the removal of the context needed only for matching the configuration structure. The removal of this context allows the rule to be more modular, since it is not then tied to a specific configuration, only to the configuration pieces used directly by the rule. The third transformation step, shown in Rule 2.3, is the replacement of variables given on the left-hand side of a rule by underscores if they are not otherwise used (in conditions or on the right-hand side). This is used to replace V' with an underscore, since it is not used elsewhere in the rule.

Since matching against lists and sets is used quite often, it is also helpful to have special notation for both lists and sets. In K, this is indicated by using "...", with a "..." at the start or end of a cell indicating a list match ("..." at the start indicates that one is matching the list tail, while "..." at the end indicates that one is matching the head), and "..." at both ends of the cell indicating a set or multiset match. In the fourth transformation step, Rule 2.4 shows the transformation of Rule 2.3 to use this notation. Finally, since some of the information in Rule 2.4 is redundant it can be removed by switching to a special rule format, where the parts of a term that are changed are underlined, with the changes to the term then placed below the underlines. The result of applying this fifth transformation to Rule 2.4 is shown in Rule 2.5. In the `k` cell, the $\cdot$ is now under $X\ \leftarrow V$, while V is now under $_$. Since both the environment and the first element (L) of the pair in *mem* do not change, each only needs to be written once.

3. The SILF Policy Framework

SILF, the Simple Imperative Language with Functions, was introduced in earlier work by the authors [Hil07, Hil09]. Here, we extend the language presented in this earlier work with a policy framework and with two policies: one for type checking programs in SILF, using type annotations; and one for checking the unit safety of programs, using type and

code annotations to indicate the units associated with program variables and values. To give an idea of the size of the specification, the policy framework is made up of 281 operators and 284 equations across 49 modules; the type checking policy of 100 equations and 17 operators across 7 modules; and the unit checking policy of 273 equations and 52 operators across 38 modules [1]. The SILF Policy Framework is available for download or online use at the SILF Policy Framework homepage [Hil].

3.1. Adding a Policy Framework to SILF

Adding a policy framework to SILF requires adding a policy-aware front end, a core abstract language semantics, generic analysis support, and the individual analysis policies. Since compatibility with existing SILF code is not an issue, to add a policy framework to SILF annotation support has been added directly to the language, versus (as was done in the C Policy Framework [Hil08] to maintain compatibility with existing C compilers) adding support through source comments. In the extended SILF syntax, type annotations are identifiers with a leading `$`, like `$int` or `$meter`. Type variables are given with similar syntax: `$$`, instead of just `$`, like `$$X`. Type identifiers and variables are used in standard type positions, like on variables and formal parameters. Code annotations are given in syntax extensions for invariants on loops (for both `while` and `for` loops), assume and assert statements, and, in function declarations, function contracts with preconditions, postconditions, and modifies (which identifies the globals changed by a function). The code annotation, an arbitrary string, is actually parsed by the policy. Each code annotation includes a *policy tag*, identifying the policy associated with the annotation language used in the annotation. This policy tag is just an identifier, and is given before the annotation, like pre(*UNITS*). This allows annotations for multiple policies to be present in the program source at once.

The core semantics includes the original SILF abstract syntax, extended with the new type and code annotation constructs mentioned above; and the configuration (i.e., K cells). The original dynamic semantics can be viewed as a special policy which ignores the additional constructs, with evaluation over a concrete, versus abstract, domain. Extensions to the semantics to support analysis include modules providing: basic logical connectives for the annotation language; pretty printing capabilities over the abstract syntax for generating error messages; support for type annotation variables with limited forms of polymorphism; additional K cells for analysis information; and operators for working with these extensions. K's modularity allows new cells to be added without requiring changes to existing rules, making it easy to extend the state with new analysis information, such as line numbers (cell *currLn*), a copy of the environment current at function entry (cell *old*), and error messages generated by the analysis (cell *log*). Many SILF language features are also given a default generic semantics, with special computation items used to indicate "hooks" whose behavior is defined by individual policies. For instance, the result of an addition expression is left up to the individual policy, since a type checking policy would have different correctness requirements than a units of measurement checking policy.

Figure 3 provides several examples of policy-generic semantics rules. Rule 3.1 is the generic rule for assignments. To check an assignment, the semantics first evaluates `X` and `E`. The computation item *checkAssign* then determines, in a policy-specific manner, whether the value of `E` can be assigned to `X` (based, for instance, on the current assigned value

[1] This actually defines three progressively more complex policies, with no single policy using all the operators, equations, or modules defined.

$$\langle k \rangle \frac{X \ := \ E}{(X, E) \curvearrowright checkAssign} ...\langle /k \rangle \tag{3.1}$$

$$\langle k \rangle \frac{X[E] := \ E'}{(X, E, E') \curvearrowright checkArrayAssign} ...\langle /k \rangle \tag{3.2}$$

$$\langle k \rangle \frac{\textsf{if } E \textsf{ then } Dt \ St \textsf{ else } Df \ Sf \textsf{ fi}}{E \curvearrowright checkIfGuard \curvearrowright if(Dt \curvearrowright St \curvearrowright Env, Df \curvearrowright Sf \curvearrowright Env)} ...\langle /k \rangle \ \langle env \rangle Env \langle /env \rangle \tag{3.3}$$

Figure 3: SILF Abstract Statement Semantics

or any annotations given on the declaration). Rule 3.2 is similar, but also evaluates the array index expression, and then uses computation item *checkArrayAssign* to ensure the assignment meets all requirements for the policy. The final rule shown, Rule 3.3, shows the generic semantics for a conditional. The conditional guard, `E`, is evaluated first; the computation item *checkIfGuard* will then check the value in a policy-specific manner. The computations stored as part of the *if* computation item will then be used to check both the then and else branches, with each computation handling the declarations and statements along that branch and restoring the environment to that active at the start of the conditional, maintaining block scoping.

The most challenging part of the generic semantics deals with handling function calls and function call sites. During analysis, each function is executed using the policy semantics. Any preconditions are first assumed correct, with postconditions verified at each function return. Since checking is static, call sites are modeled using a computation representing a summary of the called function's behavior. Any preconditions of the called function are first checked, using the actual parameter values in place of the formal parameters in the preconditions; the modifies clauses of this called function are then evaluated, generally setting any modified globals to policy-specific unknown or random values. Finally any postconditions on the called function are evaluated, with the results assumed to hold. Full details of this process can be found with the definition of the framework [Hil].

3.2. Defining A Type Checking Policy for SILF

To define a type checker for SILF as a policy, the first step is to define the analysis domain for types. The values in this domain are shown in Figure 4. Since this policy only uses type annotations, no separate code annotation language needs to be given.

```
sort BaseType .
subsort BaseType < Type .
ops $int $bool : -> BaseType .
op $array : BaseType -> Type .
op $notype : -> Type .
```

Figure 4: SILF Types Domain

The second part of defining the policy is defining the analysis-specific semantics for type checking. These rules generally follow the dynamic semantics rules closely, with the addition that error checking logic has been added to catch errors that, dynamically, led to stuck states. For instance, Figure 5 shows the original integer addition rule for the SILF dynamic semantics, Rule 3.4, as well as two typing rules for addition. The first typing rule, Rule 3.5, indicates the expected scenario: each operand is of type `$int`, with the entire operation also of type `$int`. The second, Rule 3.6, is an error case; at least one of the types is not `$int`. To handle this, the policy code generates an error message (abstracted here as *msg*) of severity 1 (an error) using `issueWarning`. It

$$i_1 + i_2 \to i, \text{ if } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{3.4}$$

$$(\$int, \$int) \curvearrowright plus \to \$int \tag{3.5}$$

$$\langle k \rangle \frac{(t, t') \curvearrowright plus}{issueWarning(1, msg) \curvearrowright \$int} ... \langle /k \rangle, \text{ if } t =/= \$int \text{ or } t' =/= \$int \tag{3.6}$$

$$\texttt{if } true \texttt{ then } Kt \texttt{ else } Kf \to Kt \tag{3.7}$$

$$\$bool \curvearrowright \texttt{if}(Kt, Kf) \to Kt \curvearrowright Kf \tag{3.8}$$

Figure 5: SILF Type Checking Policy Rules

also returns `$int` as the result type to prevent a cascade of additional errors being triggered by this one type error.

Figure 5 also provides a comparison between the rules for conditionals in the dynamic and the type checking policy semantics. Rule 3.7, part of the dynamic semantics, selects the `then` (Kt) branch in the case of a true condition (the similar else branch rule is not shown). Rule 3.8, part of the policy semantics, makes sure the condition evaluates to a boolean; after this check, the then branch and the else branch are both analyzed. Another rule, not shown, handles the case where the condition does not evaluate to `$bool`, using `issueWarning` like in Rule 3.6 to issue an error message and then, like in Rule 3.7, checking both branches of the conditional. Similar rules are used to define most of the features of the language.

```
function $int f($int x)        1
begin                          2
  return x + 1;                3
end                            4
function $int main(void)       5
begin                          6
  var $int x;                  7
  x := 3;                      8
  x := f(x);                   9
  x := f(x,x);                 10
  if x then                    11
    write 1;                   12
  fi                           13
  if (x < 5) then              14
    write 1;                   15
  else                         16
    write false;               17
  fi                           18
end                            19
```

Figure 6: Checking Types

Figure 6 shows an example of a program with type errors. Running the type checking policy over the program, the following error messages are generated:

```
ERROR on line 10: Type failure: too many arguments provided in call to function f.
ERROR on line 11: Type failure: expression x should have type $bool, but has type $int.
ERROR on line 17: Type failure: write expression false has type $bool, expected type $int.
```

In the first error, function `f` expects one parameter but is given two. In the second, the conditional expression should be a boolean, but instead an integer is provided, and unlike in languages such as C no automatic coercion is performed. In the final error message, the expression given to the write statement should be an integer, but is instead a boolean. The policy pretty printer, part of the generic analysis support defined for the framework (the basic pretty printer is actually shared between frameworks, but most of the logic is language specific), is used to generate the error messages, and is extended by the policy to correctly print the annotations.

$$(u, u) \curvearrowright plus \rightarrow u \tag{3.9}$$

$$\langle k \rangle \frac{(u, u') \curvearrowright plus}{issueWarning(1, msg) \curvearrowright \$fail} ...\langle /k \rangle, \quad \text{if } u =/= u' \text{ and } u' =/= \$cons \tag{3.10}$$

$$(u, u') \curvearrowright times \rightarrow u\ u' \tag{3.11}$$

$$V \curvearrowright \mathtt{if}(Kt, Kf) \rightarrow Kt \curvearrowright Kf \tag{3.12}$$

$$\langle k \rangle \frac{(u, u') \curvearrowright checkAssign}{\cdot} ...\langle /k \rangle, \quad \text{if } u == u' \text{ or } u' == \$cons \tag{3.13}$$

$$\langle k \rangle \frac{(u, u') \curvearrowright checkAssign}{issueWarning(1, msg)} ...\langle /k \rangle, \quad \text{if } u =/= u' \text{ and } u' =/= \$cons \tag{3.14}$$

Figure 7: SILF UNITS Policy Rules

3.3. Defining a Units Policy for SILF

The UNITS policy for SILF is similar to that defined for C in the C Policy Framework (CPF) [Hil08], and is only presented at a high level here to show the similarity to the rules for the type checking policy. The complete policy is available for download on the SILF Policy Framework site [Hil].

A program is considered unit safe if it properly follows a number of unit rules, such as only adding values with matching units. Figure 7 shows several UNITS rules. The first, Rule 3.9, is for addition, where, if both units match, the result is the same unit; Rule 3.10 is an error case for addition, where the units don't match and the second unit isn't a constant (which can be converted to any unit). Rule 3.11 is a rule for multiplication, where the resulting unit is the product of the operand units. Rules 3.12, 3.13, and 3.14 are rules for statements. Rule 3.12 handles conditionals, and is similar to Rule 3.8, except there is no need to check the value computed by the guard – any errors found in the guard expression will have already been reported, and the guard is not expected to have a specific unit (a more stringent requirement would be to enforce that the guard has no unit, but that is not done here). Rules 3.13 and 3.14 then show the regular and error cases for assignment. In Rule 3.13, the assignment is safe if the value being assigned either has the same unit or is a constant; in Rule 3.14, this condition does not hold, so an error message is issued.

4. Related Work

Many different tools and techniques have been developed around the use of annotations for program analysis. The earliest precursor to the work presented here was developed as a prototype to check the unit safety of programs written in BC [Che03]. JML [Bur03], the Java Modeling Language, provides support for code annotations, and has been used in a number of analysis tools, such as tools for runtime and static analysis. Spec# [Bar05] extends the C# language with support for code annotations and several type annotations. Eiffel [Mey88] includes direct language support for code annotations such as preconditions (require) and postconditions (ensure). None of these systems provide the same extensibility

of type annotations as seen with both the CPF and the SILF policy framework, while extensions to the provided code annotation languages (where allowed) are formalized in first-order logic.

Specifically for C, a number of annotation-based systems have been developed. LCLint [Eva94], now Splint, uses program annotations to detect potential errors in C programs, and provides limited abilities to add new annotations by allowing attributes and constraints to be defined for various C language objects. C-UNITS [Ros03], another conceptual precursor to the CPF, uses annotations to check unit safety for a limited subset of C, but is not extensible, offering no clear way to either support other analysis domains or cover unsupported features of C. Caduceus [Fil04, Fil07] provides an annotation language similar to JML; programs are verified by transforming them into a simpler language, called Why, which is then further processed to generate proof tasks for various theorem provers. Frama-C [fra] provides an extensible analysis framework, with various analyses built in OCaml as "plugins" to the core Frama-C tool. Frama-C uses the ACSL annotation language [Bau08], which is based on the annotation language used in Caduceus. To support new concepts they must be formalized in first-order logic using "logic specifications", and type annotations are not supported. Systems targeted at specific domains include VCC [Coh08], for verification of concurrent C programs, and HAVOC [Cha07], aimed at programs, such as device drivers, that perform low-level memory manipulation. CQUAL [Fos99] provides support for user-defined type annotations, referred to as type qualifiers, but cannot natively support some complex domains like units. It also does not support code annotations, such as function contracts. The current version of CPF includes new functionality over that discussed in earlier work [Hil08], including support for type annotations and `modifies` clauses and more complete support for heap-allocated values in C.

5. Conclusions

In this paper we introduced *policy frameworks*, a flexible, modular technique for adding new analysis policies and annotation languages. We presented an implemented policy framework for SILF, a simple imperative language, along with two examples of existing policies for SILF, one for types and one for units of measurement. These policies illustrate the reuse within a policy framework of the policy core, while also sharing some framework functionality with the CPF, showing that reuse across frameworks is possible as well.

In the future, we plan to extend the same concepts used here to other programming languages, potentially Java or OCaml. Of special interest is to see if it would be possible to then extend this technique to multi-language analysis (for instance, to calls from OCaml into C code). We are also working to relate the abstract semantics developed for analysis more closely with parallel efforts to develop concrete K semantics of various languages. Finally, we are investigating integrating current work on policy frameworks with work on Rascal [Kli09b, Kli09a], a language for source code analysis and transformation which should allow analysis support to be developed for significant real-world languages.

References

[Bar05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Proceedings of CASSIS'04*, *LNCS*, vol. 3362, pp. 49–69. Springer, 2005.

[Bau08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language.* 2008.

[Ber89] Jan A. Bergstra, Jan Heering, and Paul Klint. *Algebraic Specification.* ACM Press, 1989.

[Bur03] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In *Proceedings of FMICS'03*, *ENTCS*, vol. 80, pp. 75–91. 2003.

[Cha07] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A Reachability Predicate for Analyzing Low-Level Software. In *Proceedings of TACAS'07*, *LNCS*, vol. 4424, pp. 19–33. Springer, 2007.

[Che03] Feng Chen, Grigore Roşu, and Ram Prasad Venkatesan. Rule-Based Analysis of Dimensional Safety. In *Proceedings of RTA'03*, *LNCS*, vol. 2706, pp. 197–207. Springer, 2003.

[Coh08] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. A Practical Verification Methodology for Concurrent Programs, 2008.

[Eva94] David Evans, John V. Guttag, James J. Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of FSE'94*, pp. 87–96. ACM Press, 1994.

[Fil04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover Verification of C Programs. In *Proceedings of ICFEM'04*, *LNCS*, vol. 3308, pp. 15–29. Springer, 2004.

[Fil07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Proceedings of CAV'07*, *LNCS*, vol. 4590, pp. 173–177. Springer, 2007.

[Fos99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of PLDI'99*, pp. 192–203. ACM Press, 1999.

[fra] Frama-C. `http://frama-c.cea.fr`.

[Gog77] Joseph A. Goguen, James W. Thatcher, Eric G. Wagner, and Jesse Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[Gog96] Joseph A. Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996.

[Hil] Mark Hills and Grigore Roşu. SILF Policy Framework. `http://fsl.cs.uiuc.edu/index.php/SILF_Policy_Framework`.

[Hil07] Mark Hills, Traian Florin Şerbănuţă, and Grigore Roşu. A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. In *Proceedings of WRLA'06*, *ENTCS*, vol. 176, pp. 215–231. Elsevier, 2007.

[Hil08] Mark Hills, Feng Chen, and Grigore Roşu. A Rewriting Logic Approach to Static Checking of Units of Measurement in C. In *Proceedings of RULE'08.* Elsevier, 2008. To Appear.

[Hil09] Mark Hills. Memory Representations in Rewriting Logic Semantics Definitions. In *Proceedings of WRLA'08*, *ENTCS*, vol. 238(3), pp. 155–172. Elsevier, 2009.

[Kli09a] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-Programming with RASCAL. In *Proceedings of GTTSE'09*, pp. 185–238. Universidade do Minho, 2009.

[Kli09b] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, vol. 0, pp. 168–177. IEEE Computer Society, Los Alamitos, CA, USA, 2009.

[Mes04] J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of IJCAR'04*, *LNAI*, vol. 3097, pp. 1–44. Springer, 2004.

[Mes07] José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[Mey88] Bertrand Meyer. Eiffel: A Language and Environment for Software Engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[nis] The NIST Reference on Constants, Units, and Uncertainty. http://physics.nist.gov/cuu/Units/.

[Ros03] Grigore Rosu and Feng Chen. Certifying Measurement Unit Safety Policy. In *Proceedings of ASE'03*, pp. 304 – 309. IEEE, 2003.

[Ros07] Grigore Rosu. K: A Rewriting-Based Framework for Computations – Preliminary version. Tech. Rep. Department of Computer Science UIUCDCS-R-2007-2926, University of Illinois at Urbana-Champaign, 2007.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 161-176
http://rewriting.loria.fr/rta/

INFINITARY REWRITING: FOUNDATIONS REVISITED

STEFAN KAHRS

University of Kent at Canterbury, School of Computing
E-mail address: S.M.Kahrs@kent.ac.uk

Abstract. Infinitary Term Rewriting allows to express infinitary terms and infinitary reductions that converge to them. As their notion of transfinite reduction in general, and as binary relations in particular two concepts have been studied in the past: strongly and weakly convergent reductions, and in the last decade research has mostly focused around the former.

Finitary rewriting has a strong connection to the equational theory of its rule set: if the rewrite system is confluent this (implies consistency of the theory and) gives rise to a semi-decision procedure for the theory, and if the rewrite system is in addition terminating this becomes a decision procedure. This connection is the original reason for the study of these properties in rewriting.

For infinitary rewriting there is barely an established notion of an equational theory. The reason this issue is not trivial is that such a theory would need to include some form of "getting to limits", and there are different options one can pursue. These options are being looked at here, as well as several alternatives for the notion of reduction relation and their relationships to these equational theories.

1. Introduction

Infinitary rewriting deals with infinite terms, which are defined through the metric completion of finite terms through some metric. In the simplest case (metric d_∞) this is equivalent to a co-inductive definition of terms, i.e. the set of *infinitary terms* $Ter^\infty(\Sigma)$ is the *largest set* such that every t in this set has some root symbol F taken from the signature Σ and n direct subterms t_i ($1 \leq i \leq n$) that are all in $Ter^\infty(\Sigma)$, where n is the arity of F as defined by the signature. In other words, infinitary terms are defined co-inductively through the way they unfold, without a guarantee that this unfolding ever comes to an end. Infinite terms are indeed those where it does not.

Metric completion is a general-purpose semantic construction on metric spaces which "adds" to a metric space limits to all its Cauchy-sequences, in the sense that there is a dense isometric embedding of the orginal space into a complete metric space. Using metric completion with other metrics on terms than d_∞ can restrict the infinite terms under consideration (but not add others) [6], as $Ter^\infty(\Sigma)$ can also be seen as a final co-algebra. For the purposes of this paper the choice of metric is (largely) immaterial, i.e. as long it allows for infinite terms at all.

1998 ACM Subject Classification: F.4.2.

Key words and phrases: Infinitary Rewriting, Equivalence Relation, Model.

© S. Kahrs
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.161

Reductions that only involve finite terms can still approximate infinite terms "in the limit" by moving arbitrarily close (in terms of the distance function) to some such infinite term.

That is the idea behind infinitary rewriting. The problem is then to decide *how* exactly the limit is reached. For transfinite reductions we may expect something similar to metric completion, some kind of closure operation, allowing us to reach limit terms in the limit of... *what*? Here we encounter our first problem: what is the structure to which we need to add limits and what kind of limits?

Traditionally [3], the answer to this is: (i) the structure to which we add limits are (infinite) reduction sequences. As these are functions with ordinals as their domain, we can ensure that the limits are connected to "what came before" by demanding that these functions are (ii) *continuous*. The result of these choices is the notion of (weakly) convergent reduction $\twoheadrightarrow_w$. There are alternatives to continuity; we can strengthen the condition further, or weaken it by aiming for something like *adherence* rather than convergence. We are also not forced to stick to reduction sequences as our starting point: in particular, we may just as well use the reduction relation as a whole.

The initial interest in the subject of (finitary) term rewriting was triggered by a number of observations that led to the Knuth-Bendix completion procedure [9], deriving (if and when it succeeds) a decision procedure for a given equational theory. These observations establish links between the many-step rewrite relation and the equational theory. What happens to those links in infinitary rewriting?

Before we can even pose this question meaningfully we need a notion of an equational theory for infinitary rewriting. There is none in the literature, although there are ideas approximating it. One is the specific equivalence relation $\sim_{hc}$ [8] (the so-called equivalence "modulo hyper-collapsing terms"). Its construction is not easy to generalise as it relies upon being used in connection with orthogonal iTRSs: in the presence of overlapping rules the top of a hyper-collapsing term could form a redex with the context, undermining the idea that such terms are unsolvable in the sense of the λ-calculus [1]. Another idea comes from the concept of equational model [2, 3, 10], since a notion of syntactic equivalence can be derived from that: $t =_R u$ iff the equation $t = u$ holds in all equational models of R. As we shall see, the notion of equational theory derived from such classes of models is quite strong.

The reason the definition of the equational theory is an issue at all is this: on finite terms we can define the equivalence relation $=_R$ as the *smallest equivalence relation* that includes all the rules, is substitutive, and for which $F(t_1, \ldots, t_n) =_R F(u_1, \ldots, u_n)$ holds whenever $\forall i.\, t_i =_R u_i$ and F is an n-ary function symbol from the signature. But for infinitary terms this definition is not suitable, because it does not allow for any form of limit-taking: from $A =_R B$ we could not deduce that the infinite term $t = C(A, t)$ is $=_R$-equivalent to the infinite term $u = C(B, u)$, since the equivalence closure only permits a *finite number* of equation applications. For the same reason, the transfinite reduction relation $\twoheadrightarrow$ (in any of the variations we consider) would not be a subrelation of $=_R$.

Thus some form of limit-taking needs to be incorporated into a suitable notion of equational theory. There are several ways in which one can allow for limit-taking that lead to different notions of equivalence:

- the conservative approach: form the equivalence closure of $\twoheadrightarrow$, for any given notion of transfinite reduction relation $\twoheadrightarrow$;

- defining an equivalence relation: take the *blue* and equivalence closure of the single-step rewrite relation $\to$, where "*blue*" is a placeholder for properties that ensure that certain limits are included in the relation.
- inducing an equivalence from a class of models

The conservative approach is pragmatic and makes the connection to the reduction relation straightforward, but it denies rules true equational status. For example, it does not permit us to form an infinite sequence by applying rules forwards as well as backwards and conclude that the limit of the sequence is equivalent to its beginning.

2. Preliminaries

We need several notions from Topology and Infinitary Rewriting. This section contains those definitions, to make the paper self-contained.

A property P on subsets of a set A is said to be *closable* if the intersection $\bigcap_{i\in I} A_i$ of any family of subsets $A_i \subseteq A, i \in I$ that each satisfy P itself satisfies the property. As I may be empty, A has to satisfy P too. If P is closable we can form the P-closure of any subset $K \subset A$, the smallest subset of A that contains K and satisfies P. In particular, this concept will be used on relations, viewing them as sets of pairs. Clearly, the conjunction of closable properties is closable.

2.1. Topology

A *topological space* is a pair $(S, \mathcal{O})$ where S is a set and $\mathcal{O}$ a subset of $\wp(S)$ such that it is closed under finite intersections and arbitrary unions, and $\emptyset, S \in \mathcal{O}$. The elements of $\mathcal{O}$ are called *open* sets, their complements w.r.t. S are called *closed* sets. The *closure* of a subset $A \subset S$ is the intersection of all closed sets that contain A, and is written $\mathrm{Cl}(A)$. A *neighbourhood* of $x \in S$ is a set $N \subseteq S$ for which there is an $A \in \mathcal{O}$ such that $x \in A \subseteq N$. A point $z \in S$ is called *discrete* iff $\{z\}$ is open.

A topological space S is called T_0 iff $\forall x, y \in S.\, x \in \mathrm{Cl}(\{y\}) \wedge y \in \mathrm{Cl}(\{x\}) \Rightarrow x = y$. It is called T_1 iff all singleton sets are closed. It is T_2 (or Hausdorff) iff any two distinct points in S have disjoint neighbourhoods.

A function $f : A \to B$ between topological spaces is called continuous iff $f^{-1}(X)$ is open whenever X is open. A function $f : A \to B$ between topological spaces is an *open map* iff it preserves open sets; it is a *closed map* iff it preserves closed sets. A relation R between topological spaces A and B is called *lower semi-continuous* (or lsc) if R^{-1} preserves open sets, and *upper semi-continuous* (or usc) if R^{-1} preserves closed sets.

A subset F of a topological space is called *compact* iff whenever $\bigcup_{i\in I} A_i \supseteq F$, where each A_i is open, then there is a finite subset $J \subseteq I$ such that $\bigcup_{i\in J} A_i \supseteq F$.

A metric space is a set (M, d) where M is a set and $d : M \times M \to \mathcal{R}$ a distance function such that for all $x, y, z \in M$: (i) $d(x, y) = 0 \iff x = y$ and (ii) $d(x, z) \leq d(x, y) + d(z, y)$. For an ultra-metric space (ii) is replaced by the stronger condition $d(x, z) \leq \max(d(x, y), d(z, y))$. The topology of a metric space is defined as follows: $A \subseteq M$ is open iff $\forall x \in A.\, \exists \epsilon > 0. \forall y \in M.\, d(x, y) < \epsilon \Rightarrow y \in A$.

The metric completion of (M, d) is the unique (up to isomorphism) metric space $(M^\bullet, d^\bullet)$, with a function $e : M \to M^\bullet$, such that e preserves distances and $\mathrm{Cl}(e(M)) = M^\bullet$.

A function between metric spaces A and B is *uniformly continuous* iff $\forall.\, \epsilon > 0.\, \exists \delta > 0.\, \forall x, x' \in A.\, d_A(x, x') < \delta \Rightarrow d_B(f(x), f(x')) < \epsilon$. Uniformly continuous functions have

unique continuous extensions to the respective metric completions. A special case are *non-expansive* functions where $\delta = \epsilon$.

2.2. Infinitary Term Rewriting

A *signature* is a pair $\Sigma = (\mathcal{F}, \#)$ where $\mathcal{F}$ is a set (of function symbols) and $\# : \mathcal{F} \to \mathcal{N}$ is the function assigning each symbol its arity. We assume an infinite set *Var* of *variables*, disjoint from $\mathcal{F}$. The set of *finite terms* over Σ is called $Ter(\Sigma)$ and it is defined to the smallest set such that (i) $Var \subset Ter(\Sigma)$ and (ii) $F(t_1, \ldots, t_n) \in Ter(\Sigma)$ whenever $F \in \mathcal{F} \wedge \#(F) = n \wedge \{t_1, \ldots, t_n\} \subset Ter(\Sigma)$. The *root symbol* of a term $F(t_1, \ldots, t_n)$ is F, the root symbol of a variable x is x.

A Σ-algebra is a set A together with functions $F_A : A^n \to A$ for every $F \in \mathcal{F}$ with $\#(F) = n$. A *valuation* into A is a function $\rho : Var \to A$. Any Σ-algebra A determines an interpretation function $[\![_]\!]_A : Ter(\Sigma) \times (Var \to A) \to A$ as follows:

$$\begin{array}{rcll} [\![x]\!]^\rho_A & = & \rho(x), & \text{if } x \in Var \\ [\![F(t_1, \ldots, t_n)]\!]^\rho_A & = & F_A([\![t_1]\!]^\rho_A, \ldots, [\![t_n]\!]^\rho_A) & \end{array}$$

Infinitary terms are defined through a metric completion process. For this paper we focus on the metric d_∞ which is defined as follows inductively on finite terms: $d_\infty(t, u) = 1$ iff t and u have different roots; $d_\infty(t, t) = 0$; otherwise, $d_\infty(F(t_1, \ldots, t_n), F(u_1, \ldots, u_n)) = 1/2 * \max_{1 \leq i \leq n} d_\infty(t_i, u_i)$. $(Ter(\Sigma), d_\infty)$ is an ultra-metric space and we write $(Ter^\infty(\Sigma), d_\infty)$ for its metric completion, which is also a Σ-algebra [6]. There is a more general notion of *term metric* m from which distance functions d_m and the corresponding metric completions $Ter^m(\Sigma)$ can be derived [6].

A rewrite rule is pair (l, r), usually written $l \to r$, such that $l \in Ter(\Sigma) \setminus Var$, $r \in Ter^\infty(\Sigma)$ and all variables occurring in r also occur in l.

An iTRS is a pair (Σ, R) where Σ is a signature and R a set of rewrite rules for that signature. The rewrite step relation $\to_R$ relates $C[\sigma(l)] \to_R C[\sigma(r)]$, where $l \to r$ is a rewrite rule in R, σ a substitution, and $C[\,]$ a context. Substitution application and context application are uniquely derived from their respective concepts on finite terms [6], and can also be defined in terms of the $[\![_]\!]_{Ter^\infty(\Sigma)}$ interpretation.

3. Transfinite Sequences

In the following we are looking at several notions of transfinite reduction relations $X(\to_R)$. These are all functions of the single step reduction relation $\to_R$. We call $X(\to_R)$ *infinitarily transitive* if $X(\to_R) = X(X(\to_R))$, i.e. if it is a fixpoint of X. Usually, we take $\to_R$ to be clear from the context and write $\twoheadrightarrow_x$ for $X(\to_R)$ and $\twoheadrightarrow_{xx}$ for $X(X(\to_R))$. We also write $\twoheadleftarrow_x$ for $\twoheadrightarrow_x^{-1}$, $\downarrow_x$ for $\twoheadrightarrow_x ; \twoheadleftarrow_x$, and $\uparrow_x$ for $\twoheadleftarrow_x ; \twoheadrightarrow_x$.

The reason the property "infinitarily transitive" is desirable is similar to wanting that $\to^*_R$ is the same as $(\to^*_R)^*$; the property also features in the proofs of [7].

Transfinite sequences of terms can be defined as functions from an ordinal (the index domain) to the set of (infinitary) terms, viewing ordinals as von Neumann ordinals, i.e. the ordinal α is the set of all ordinals strictly smaller than α. Reduction sequences of an iTRS are those where neighbouring elements are within the single-step reduction relation, i.e. if $f : \alpha \to Ter^\infty(\Sigma)$ is our reduction sequence then $f(n) \to_R f(n+1)$, provided $n + 1 < \alpha$.

This works fine for *finite* sequences. For infinite sequences this definition fails to put any constraints whatsoever what happens at $f(\lambda)$, for limit ordinals λ.

3.1. Standard solution: weak convergence

The traditional choice to fix this is to demand that the function f is continuous w.r.t. the usual order topology. Effectively, this means that $f(\lambda)$ must be the (unique) limit of $f(\gamma)$, as γ approaches λ from below. To express it in terms of distances:

$$\forall \epsilon > 0.\, \exists \gamma.\, \forall \gamma'.\, \gamma \leq \gamma' < \lambda \Rightarrow d_\infty(f(\gamma'), f(\lambda)) < \epsilon$$

If the indexing set of a transfinite reduction sequence is a successor ordinal then we have a *closed sequence*, because it is a sequence with a last element: if $\alpha+1$ is the domain of f then the last element of the sequence is $f(\alpha)$.

This also gives us a way of defining the relation $\twoheadrightarrow_w$: $t \twoheadrightarrow_w u$ iff there is some ordinal α and some closed reduction sequence $f : \alpha+1 \to \mathit{Ter}^\infty(\Sigma)$ such that $f(0) = t$ and $f(\alpha) = u$.

In the infinitary rewriting literature this is often called "weak convergence" [8], where "strong convergence" requires that the sequences converges solely due to the positioning of redexes, i.e. if $f(\gamma) = C_\gamma[\sigma_\gamma(l_\gamma)] \to_R f(\gamma+1) = C_\gamma[\sigma_\gamma(r_\gamma)]$ we can build a new sequence $g(\gamma) = C_\gamma[x]$ (i.e. replacing all contracted redexes with the variable x); f is then strongly converging if g converges too, and to the same limit as f. Expressing strong convergence as a function of $\to_R$ (rather than R) would require to recover a minimal rule set from the relation $\to_R$ — a slightly delicate issue that goes beyond the scope of this paper.

Example 3.1. Consider the iTRS with rules

$$\begin{array}{rcl} F(A,x) & \to & F(B,D(x)) \\ B & \to & A \end{array}$$

We have $F(A,x) \to_R^2 F(A,D(x))$, but we do not have $F(A,x) \twoheadrightarrow_w F(A,D^\infty)$, because A would change to B at every other step. If we add the rule $F(A,x) \to F(A,D(x))$ to the system then $\to_R^*$ does not change, but $\twoheadrightarrow_w$ would change and now include $F(A,x) \twoheadrightarrow_w F(A,D^\infty)$.

Proposition 3.2. *$\twoheadrightarrow_w$ is in general not infinitarily transitive (though it is on converging iTRSs* [7]*).*

Proof. See example 3.1. Since $F(A,x) \to_R^2 F(A,D(x))$ we also have $F(A,x) \twoheadrightarrow_w F(A,D(x))$ and consequently $F(A,x) \twoheadrightarrow_{ww} F(A,D^\infty)$. ∎

Incidentally, this contradicts theorem 1(c) in [2].

3.2. Adherence

Instead of asking for convergence we can ask for adherence: $t \twoheadrightarrow_a u$ is defined like $t \twoheadrightarrow_w u$, except for one thing: instead of requiring that the witnessing indexing function f is continuous at limit ordinals λ we require that it is "adherent": This is in a certain sense a concept dual to convergence, because instead of demanding that a sequence is eventually always within a neighbourhood, the definition asks instead that it always eventually goes there. Formally:

$$\forall \epsilon > 0.\, \forall \gamma < \lambda.\, \exists \gamma'.\, \gamma \leq \gamma' < \lambda \wedge d_\infty(f(\gamma'), f(\lambda)) < \epsilon$$

The difference is that adherence merely requires that (any neighbourhood of) an accumulation point is visited by the sequence for index positions arbitrarily close to λ, without demanding that the sequence stays there. Intuitively, adherence requires that a cofinal subsequence of a reduction sequence converges to the limit, allowing for other terms in the sequence as computational noise.

The result of this is that a sequence can adhere to more than one limit. Certainly, any sequence converging to a limit adheres to it and therefore:

Proposition 3.3. $\twoheadrightarrow_w \subseteq \twoheadrightarrow_a$

Clearly, $\twoheadrightarrow_a$ is (finitary) transitive, $\twoheadrightarrow_a;\twoheadrightarrow_a \subseteq \twoheadrightarrow_a$, because the adherence condition never stops adherent sequences from being concatenated. One peculiarity of adherence over convergence is that the notion is less sensitive to the notion of the single-step relation, in the sense that if $\to_R^* = \to_S^*$ then the adherence relations of $\to_R$ and $\to_S$ are identical too. Also:

Proposition 3.4. $\twoheadrightarrow_a$ *is infinitarily transitive.*

Proof. Let $f : \alpha + 1 \to \mathit{Ter}^\infty(\Sigma)$ be the sequence witnessing $t \twoheadrightarrow_{aa} u$. We need to show $t \twoheadrightarrow_a u$. This can be proved by induction on α.

If $\alpha = 0$ then $t = u$ and the result follows by reflexivity.

If $\alpha = \beta + 1$ then there is a t' such that $t \twoheadrightarrow_{aa} t' \twoheadrightarrow_a u$, where the sequence witnessing $t \twoheadrightarrow_{aa} t'$ has length β. By induction hypothesis $t \twoheadrightarrow_a t'$, and the result follows by (finitary) transitivity of $\twoheadrightarrow_a$.

If α is a limit ordinal then the restriction of f to domain α has a cofinal subsequence [7] $g : \beta \to \alpha$ such that $f \circ g$ converges to $f(\alpha)$. Once we expand every $\twoheadrightarrow_a$ step in f to a new sequence h we have that $f = h \circ g'$ for some cofinal subsequence g' of h and thus $f \circ g = (h \circ g') \circ g = h \circ (g' \circ g)$ where $g' \circ g$ is a cofinal subsequence of h. ∎

By implication proposition 3.4 also shows that the inclusion $\twoheadrightarrow_w \subset \twoheadrightarrow_a$ is (in general) proper, as witnessed by example 3.1.

Adherence can also be characterised as follows: let W be the function mapping a relation $\to_R$ to its weak convergence relation $\twoheadrightarrow_w$. Then $\twoheadrightarrow_a$ is the least fixpoint of W that contains the single-step relation.

4. Relations

Instead of completing sequences by adding limits or accumulation points, we can define $\twoheadrightarrow$ more directly through closable properties of relations. There are the following notions of interest:

4.1. Pointwise Closure

We can view relations as set-valued functions, and add limits to their range. This leads to the following concept:

Definition 4.1. A relation R between topological spaces is called *pointwise closed* iff the sets $R^x = \{y \mid x\, R\, y\}$ are all closed.

Proposition 4.2. *Being pointwise closed is a closable property of relations.*

Proof. Let $A = \bigcap_i R_i$. Then $A^x = \{y \mid x\, A\, y\} = \{y \mid \forall i.\ x\, R_i\, y\} = \bigcap_i R_i^x$. Hence A^x is an intersection of closed sets and therefore closed. ∎

This allows us to use pointwise closure as a relation-constructing property.

Definition 4.3. The relation $P(\to_R) = \twoheadrightarrow_p$ is defined as the smallest reflexive, transitive and pointwise closed relation containing $\to_R$.

That $\twoheadrightarrow_p$ is infinitarily transitive is trivial by construction. We can explain $t \twoheadrightarrow_p u$ as "t can rewrite to something arbitrarily close to u", but if we want to get any closer we may have to start all over again from t.

Proposition 4.4. $\twoheadrightarrow_a \subseteq \twoheadrightarrow_p$.

Proof. By induction on the indexing ordinals for the sequences witnessing $t \twoheadrightarrow_a u$. The interesting case for $f : \alpha + 1 \to \mathit{Ter}^\infty(\Sigma)$ is when α is a limit ordinal. By induction hypothesis, $f(0) \twoheadrightarrow_p f(\gamma)$, for all $\gamma < \alpha$. The restriction of f to α has to contain a subsequence that converges to $f(\alpha)$. But then $f(\alpha)$ has to be in the closure of the $f(\gamma)$ and as $\twoheadrightarrow_p$ is pointwise closed the result follows. ∎

However, the relations $\twoheadrightarrow_a$ and $\twoheadrightarrow_p$ are not always the same:

Example 4.5.

$$\begin{array}{rcl} A & \to & B(A) \\ A & \to & C \\ B(C) & \to & D(C) \\ B(D(x)) & \to & D(D(x)) \end{array}$$

In this system we have $A \to_R^* D^n(C)$ for any finite n and therefore $A \twoheadrightarrow_p D^\infty$. But there is no single adherent (or convergent) sequence that can build up to that limit; as soon as a D appears in a reduct of A the reduction sequence is guaranteed to terminate.

Usually we can construct $\twoheadrightarrow_p$ more directly, as the pointwise closure of $\to_R^*$ (call it $\twoheadrightarrow_{p0}$).

Theorem 4.6. *If $\to_R$ is lsc then $\twoheadrightarrow_p = \twoheadrightarrow_{p0}$.*

Proof. It suffices to show that $\twoheadrightarrow_{p0}$ is transitive. Since $\to_R$ is lsc so is $\to_R^*$ [6]. Suppose $A \twoheadrightarrow_{p0} B \twoheadrightarrow_{p0} C$.

We need to show $A \twoheadrightarrow_{p0} C$, which means that for any $\epsilon > 0$ there is a C_ϵ such that $A \to_R^* C_\epsilon$ and $d_\infty(C_\epsilon, C) < \epsilon$. Because $\to_R^*$ is lsc and $B \to_R^* C$ for every $\epsilon > 0$ there is a $\delta > 0$ such that for all B' with $d_\infty(B', B) < \delta$ there is a $C_\epsilon(B')$ with $B' \to_R^* C_\epsilon(B')$ and $d_\infty(C_\epsilon(B'), C) < \epsilon$. Since $A \twoheadrightarrow_{p0} B$ we can find B_δ with $d_\infty(B_\delta, B) < \delta$ and $A \to_R^* B_\delta$. Hence $A \to_R^* C_\epsilon(B_\delta)$. ∎

In [6] a number of conditions are given under which the relation $\to_R$ is *uniformly lsc*, for a variety of term metrics. For the much weaker condition that $\to_R$ is lsc it suffices to require that the rules are left-linear, and in that case this is even independent of the term metric.

4.2. Topological Closure

The pointwise closure add limits to a relation at the "result side", and stays in this respect still very much within the intuition behind infinitary rewriting. Going beyond that and allowing the input side to change as well leads to fairly unintuitive relations.

For example, another closable property on relations between topological spaces A and B is that their set of pairs (their graph) is closed in the product space $A \times B$.

Definition 4.7. The relation $C(\to_R) = \twoheadrightarrow_t$ is the smallest reflexive and transitive relation containing $\to_R$ such that its graph is closed.

This means: if t_n and u_n are sequences converging to t and u, respectively, and if for all i: $t_i \twoheadrightarrow_t u_i$, then $t \twoheadrightarrow_t u$.

Clearly, closed relations are also pointwise closed and therefore $\twoheadrightarrow_p \subseteq \twoheadrightarrow_t$. Again, the inclusion is proper:

Example 4.8.

$$\begin{array}{rcl} \mathrm{LEQ}(0,x) & \to & T \\ \mathrm{LEQ}(S(x),0) & \to & F \\ \mathrm{LEQ}(S(x),S(y)) & \to & \mathrm{LEQ}(x,y) \end{array}$$

The infinite term $t = \mathrm{LEQ}(S^\infty, S^\infty)$ only reduces to itself, in a single step, and thus also $\forall u.\ t \twoheadrightarrow_p u \Rightarrow t = u$. But we also have $t \twoheadrightarrow_t T$ and $t \twoheadrightarrow_t F$, because the sequences $a_n = LEQ(S^n(0), S^n(0))$ and $b_n = LEQ(S^{n+1}(0), S^n(0))$ both converge to t, but $a_n \twoheadrightarrow_t T$ and $b_n \twoheadrightarrow_t F$.

In contrast to $\twoheadrightarrow_p$ we usually cannot construct $\twoheadrightarrow_t$ as the topological closure of $\to_R^*$ (call it $\twoheadrightarrow_{t0}$), because that relation is often not transitive:

Example 4.9. Add the rule $\mathrm{INF} \to S(\mathrm{INF})$ to example 4.8. Then $\mathrm{LEQ}(\mathrm{INF},\mathrm{INF}) \twoheadrightarrow_{t0} \mathrm{LEQ}(S^\infty, S^\infty) \twoheadrightarrow_{t0} T$, but we do not have $\mathrm{LEQ}(\mathrm{INF},\mathrm{INF}) \twoheadrightarrow_{t0} T$.

5. Notions of Equivalence

When using rewrite relations to (semi-)decide an equivalence we want that equivalent terms have common reducts. Hence:

Definition 5.1. A pre-order $\twoheadrightarrow_x$ is called a *semi-decider* for an equivalence $=_E$ iff (i) $\twoheadrightarrow_x \subseteq =_E$ and (ii) $=_E \subseteq \downarrow_x$.

The first condition gives us soundness (if terms have common reducts they are equivalent), the second completeness. If $=_E$ is the equivalence closure of $\twoheadrightarrow_x$ then (i) is trivial and (ii) is equivalent to infinitary confluence, $\uparrow_x \subseteq \downarrow_x$. However, for infinitary rewriting this is a big "if".

In the presence of infinite terms, ordinary congruence relations fail to capture what is needed for equational reasoning in infinitary rewriting as equivalence closure is an inductive concept, not a coinductive one. This problem shows up in two separate ways: (i) for including transfinite reductions in the equivalence, and (ii) for allowing infinitely many subterm changes in a term of infinite size.

However, any equivalence relation $\sim$ on a topological space A induces a canonical topology on the quotient $A/{\sim}$: a set of equivalence classes is open iff their union is open in the topology of A. This condition is the finest topology that makes the projection map $[_]_\sim : A \to A/{\sim}$ continuous. This also means that if we have *any* converging sequence $f(n)$ in A then $[f(n)]_\sim$ is converging in $A/{\sim}$.

Example 5.2. Consider the iTRS with the single rule $C \to S(C)$. Take as equivalence $\approx$ the congruence closure of the equation $C = S(C)$. The reduction sequence $C \to S(C) \to S(S(C)) \to \ldots$ converges to S^∞. By continuity, the sequence $[C]_\approx, [S(C)]_\approx, [S(S(C))]_\approx, \ldots$ converges to $[S^\infty]_\approx$. However, $[C]_\approx = [S(C)]_\sim = [S(S(C))]_\approx, \ldots$ and $[C]_\approx \neq [S^\infty]_\approx$.

Example 5.2 shows that quotient spaces can have very poor separation properties, e.g. in the example $Ter^\infty(\Sigma)/\approx$ is not T_1. These separation properties closely correspond to properties of equivalence relations, in the sense that they indirectly provide recipes for adding limits to an equivalence. This leads to the following concepts:

Definition 5.3. An equivalence relation $\sim$ on a topological space A is called *weakly separating*, iff:

$$\forall x, y \in A.\, x \in \mathrm{Cl}([y]_\sim) \wedge y \in \mathrm{Cl}([x]_\sim) \Rightarrow x \sim y$$

Proposition 5.4. *$A/\sim$ is a T_0 space iff $\sim$ is weakly separating.*

Proof. Let $\sim$ be weakly separating and $[x]_\sim$ and $[y]_\sim$ be accumulation points of each other. Then $[x]_\sim \in \mathrm{Cl}(\{[y]_\sim\})$ which is equivalent to $[x]_\sim \subseteq \mathrm{Cl}([y]_\sim)$, hence $x \in \mathrm{Cl}([y]_\sim)$; the same argument gives $y \in \mathrm{Cl}([x]_\sim)$. As $\sim$ is weakly separating $x \sim y$ and so $[x]_\sim = [y]_\sim$.

If $\sim$ is not weakly separating then any witnessing counterexample is also a counterexample against $A/\sim$ being T_0. ∎

Given any relation R, we can form the "weakly separating equivalence closure" due to the following property:

Proposition 5.5. *Being weakly separating is a closable property.*

Proof. Clearly, the intersection $\sim = \bigcap_i \sim_i$ gives another equivalence where each equivalence class $[a]_\sim$ is the intersection of the equivalence classes $[a]_{\sim_i}$. Now assume $x \in \mathrm{Cl}([y]_\sim)$ and $y \in \mathrm{Cl}([x]_\sim)$. $x \in \mathrm{Cl}([y]_\sim) = \mathrm{Cl}(\bigcap_i [y]_{\sim_i}) \subseteq \bigcap_i \mathrm{Cl}([y]_{\sim_i})$. Hence, for all i, $x \in \mathrm{Cl}([y]_{\sim_i})$, and by the dual argument $y \in \mathrm{Cl}([x]_{\sim_i})$. Since each $\sim_i$ is weakly separating this implies $x \sim_i y$, for all i, and so $x \sim y$. ∎

T_0 is a very weak form of separation and we do not have that $\twoheadrightarrow_w$ would be included in the weakly separating equivalence closure of its rules. Often, the weakly separating equivalence closure makes no difference, but there are cases where it does:

Example 5.6. Consider the following specification of equality and logical negation:

$$\begin{aligned}
E(x, x) &= T \\
E(0, S(x)) &= F \\
E(S(x), 0) &= F \\
E(S(x), S(y)) &= E(x, y) \\
N(T) &= F \\
N(F) &= T
\end{aligned}$$

Instantiating the first equation we have $E(S^\infty, S^\infty) = T$. We can also derive $E(S^n(0), S^\infty) = F$, for any finite n. Thus $[T]_\sim$ is in the closure of $[F]_\sim$, where $\sim$ is the equivalence closure of $\to_R$. Moreover, for any finite n, $N(E(S^n(0), S^\infty)) = N(F) = T$; hence the closure of $[T]_\sim$ will also contain $N(E(S^\infty, S^\infty)) = N(T) = F$. Therefore, the weakly separating closure of $\sim$ will relate T and F.

Without the first equation the weakly separating closure would not change the relation: $E(S^\infty, S^\infty)$ would be in an equivalence class of its own; both the closures of $[T]_\sim$ and $[F]_\sim$ would contain that class, but not vice versa.

Definition 5.7. An equivalence relation $\sim$ on a topological space A is called *separating* iff all its equivalence classes are closed.

Again, we can use this concept as a closure principle:

Proposition 5.8. *Being separating is a closable property.*

Proof. The equivalence classes of $\sim$ are closed iff $\sim$ is pointwise closed, hence this follows from proposition 4.2. ∎

Example 5.9. If we remove the first equation $E(x, x) = T$ from example 5.6 then T and F would not be related by the weakly separationg closure of $\sim$, but they would be by the separating closure: since $E(S^\infty, S^\infty)$ is in the closure of both $[T]_\sim$ and $[F]_\sim$, T, F and $E(S^\infty, S^\infty)$ would all be equivalent under the separating closure of $\sim$.

Thus "separating" is a much stronger property than "weakly separating". If an iTRS (over metric d_∞) contains a collapsing rule $C[x, \ldots, x] \to x$ then the sequence $x_0 = x$, $x_{n+1} = C[x_n, \ldots, x_n]$ converges to a limit C^∞. The same is true if we start the sequence with $x_0 = y$ instead. Hence, $x \sim y$ if $\sim$ is the separating closure of $\to_R$. For other metrics d_m collapsing rules may not cause that problem, as the sequence x_n could be diverging under d_m.

In finitary term rewriting, the derivability of $x =_R y$ is used as the standard criterion for inconsistency. For infinitary rewriting (over metric d_∞) this becomes trivial for separating equivalences: the separating closure of $\to_R$ contains the pair (x, y) iff R contains a collapsing rule. The reason: if it does contain a collapsing rule then the previous argument applies, if it does not then each set $\{x\}$ remains an equivalence class of its own, since x is a discrete point in $Ter^\infty(\Sigma)$: thus $\{x\}$ is closed and no set not containing x has it in its closure. In particular, even example 5.6 is consistent despite $T \sim F$.

Separating equivalences characterise T_1 spaces:

Proposition 5.10. *$A/\sim$ is T_1 iff $\sim$ is a separating equivalence.*

Proof. Folklore[5, p. 207]. ∎

In a T_1 space, a sequence that is eventually constant can only converge to that constant, because all singleton sets in a T_1 space are closed. That also means that if all elements of a converging sequence are equivalent to each other then that also applies to the limit. More generally:

Theorem 5.11. *Let $\sim$ be the separating equivalence closure of $\to_R$. Then $\twoheadrightarrow_p \subseteq \sim$.*

Proof. Immediate, because $\sim$ is pointwise closed, reflexive, transitive, and contains $\to_R$, and $\twoheadrightarrow_p$ is by definition the smallest such relation. ∎

Theorem 5.12. *A pre-order $\twoheadrightarrow_x$ is a semi-decider for the separating closure of $\twoheadrightarrow_x$ iff (i) $\uparrow_x \subseteq \downarrow_x$ and (ii) $\downarrow_x$ is pointwise closed.*

Proof. Let $=_x$ be the separating closure of $\twoheadrightarrow_x$. The "only if" part of the theorem is trivial.

For the "if" part, $=_x$ can be computed by repeatedly (and alternatingly) applying the equivalence closure and pointwise closure, starting with $\twoheadrightarrow_x$. The result can be shown by induction on the number of closures needed to derive a particular equation $t =_x u$.

Suppose $t =_x u$ is derived from an equivalence closure with $t = t_1 =_x t_2 =_x, \ldots, =_x t_n = u$, where each equation $t_i =_x t_{i+1}$ has a shorter derivation. Then by induction hypothesis $t_i \downarrow_x t_{i+1}$, and by repeatedly applying condition (i) we have $t \downarrow_x u$.

Alternatively $t =_x u$ is derived from the pointwise closure, i.e. u is in the closure of the set of all t_i with $t =_x t_i$ and where these equations have shorter derivations. Then, for all i, $t \downarrow_x t_i$ by induction hypothesis and $t \downarrow_x u$ by condition (ii). ∎

Although trivial in the proof, the "only if" part of the theorem shows that confluence alone (condition (i)) is insufficient to make $\twoheadrightarrow_x$ a semi-decider. Confluence properties of $\twoheadrightarrow_x$ aside (for $\twoheadrightarrow_a$ and $\twoheadrightarrow_p$ this is entirely new territory), when is its joinability relation pointwise closed? A sufficient condition is the following:

Theorem 5.13. *If $\twoheadrightarrow_x$ is pointwise closed and usc then $\downarrow_x$ is pointwise closed.*

Proof. Let $t \twoheadrightarrow_x a_i \twoheadleftarrow_x u_i$ with $i \in I$ for some index set I. Let $u \in \mathrm{Cl}(\{u_i \mid i \in I\})$.

Let $A = \mathrm{Cl}(\{a_i \mid i \in I\})$ then $t \twoheadrightarrow_x a$ for any $a \in A$ since $\twoheadrightarrow_x$ is pointwise closed. The set $\twoheadrightarrow_x^{-1}(A)$ clearly contains all the u_i and as $\twoheadrightarrow_x$ is usc it must be a closed set, so it does contain u too. Hence $u \twoheadrightarrow_x a'$ for some $a' \in A$ and thus $t \downarrow_x u$. ∎

In the previous examples in this section the separating closure of $\rightarrow_R$ was not just pointwise closed but also closed. While this is often the case there are exceptions, which means that we do not have that $\twoheadrightarrow_t$ is always included in the separating closure of $\rightarrow_R$; for example, if the original equivalence just contained $F(A^n(x)) \sim F(B^n(x))$ for all n then the separating closure would not add $F(A^\infty) \sim F(B^\infty)$, so it would not be closed.

Definition 5.14. An equivalence relation $\sim$ on a topological space A is called *strongly separating* iff its graph is closed.

Proposition 5.15. *Being strongly separating is a closable property.*

Proof. Trivial, as any intersection of closed sets is closed. ∎

Equally trivially, the strongly separating closure of $\rightarrow_R$ contains $\twoheadrightarrow_t$, by the analogous argument to theorem 5.11. Strongly separating equivalences (almost) characterise Hausdorff spaces:

Proposition 5.16. *If $A/\!\sim$ is T_2 then $\sim$ is strongly separating. If $\sim$ is strongly separating and $[_]_\sim$ is an open map then $A/\!\sim$ is T_2. Or: if A is compact and $\sim$ is strongly separating then $A/\!\sim$ is T_2.*

Again, these are well-established results in topology. The first part of Proposition 5.16 can influence our choice for the class of algebraic models: if all our algebraic models are Hausdorff (e.g. metric spaces) then their induced equational theory is automatically strongly separating. Regarding the second part, open maps are functions that map open sets to open sets. For infinitary rewriting, the map $[_]_\sim$ (where $\sim$ is the strongly separating equivalence closure of $\rightarrow_R$) is typically not open though, e.g.:

Example 5.17. Take the iTRS with rule: $F(x) \rightarrow G(x, x)$. Take as open set an ϵ-ball around $F(S^\infty)$. Let $\sim$ be the strongly separating equivalence closure of $\rightarrow_R$. The union of the $\sim$-equivalence classes of all terms in the ϵ-ball contains $G(S^\infty, S^\infty)$; it is not open, because it does not contain $G(S^\infty, S^n(x))$ for any finite n.

However, one can argue that strongly separating equivalence characterise T_2 separation for infinitary rewriting in most cases, because of the third part of Proposition 5.16 and the following:

Theorem 5.18. *If the signature Σ is finite the set of ground terms of $Ter^\infty(\Sigma)$ is compact.*

Proof. Let $f : \omega \to Ter^\infty(\Sigma)$ be a sequence of infinitary ground terms. Because Σ is finite at least one function symbol F occurs infinitely often as root symbol in f. Within the infinite subsequence of f that always has F as root, infinitely many times at least one function symbol G will occur infinitely many times in the first argument position. Iterating this argument leads to an accumulation point of f which is a ground term. ∎

This argument fails for any term metric m such that $Ter^m(\Sigma)$ is not homeomorphic to $Ter^\infty(\Sigma)$, because sequences of finite ground terms that converge under d_∞ but not under d_m have no accumulation point in $Ter^m(\Sigma)$.

Without the finiteness of Σ the pigeon-hole principle in the proof fails. The restriction to ground terms is necessary as there are infinitely many variables: take the associative and commutative theory of a binary function symbol G, then the equivalence class of $G(x_1, G(x_2, \ldots))$ is not compact.

Theorem 5.19. *A pre-order $\twoheadrightarrow_x$ is a semi-decider for the strongly separating closure of $\twoheadrightarrow_x$ iff (i) $\uparrow_x \subseteq \downarrow_x$ and (ii) $\downarrow_x$ is closed.*

Proof. Analogous to Theorem 5.12. ∎

Again, this raises the issue under which conditions $\downarrow_x$ is closed.

Theorem 5.20. *If $\twoheadrightarrow_x$ is closed and usc then $\downarrow_x$ is closed.*

Proof. Let $a_i \twoheadrightarrow_x c_i \twoheadleftarrow_x b_i$ for all $i \in \omega$ and let a and b limits of the sequences a_n and b_n, respectively. We need to show $a \downarrow_x b$.

Let $C = \mathrm{Cl}(\{c_i \mid i \in \omega\})$ and $A = \{a' \mid a \twoheadrightarrow_x a'\}$. A is closed because $\twoheadrightarrow_x$ is a closed relation, and therefore $C' = C \cap A$ is closed too. Because $\twoheadrightarrow_x$ is usc the set $\twoheadrightarrow_x^{-1}(C)$ must contain a and thus C' is non-empty. If C' contains c_i for infinitely many i then $\twoheadrightarrow_x^{-1}(C')$ also contains the corresponding b_i and (by the usc property) their limit b, hence $a \downarrow_x b$. Otherwise, we can w.l.o.g. assume that all elements $c \in C'$ arise as limits of subsequences of c_n and thus $b \twoheadrightarrow_x c$ for any such c by closedness of $\twoheadrightarrow_x$, and thus again $a \downarrow_x b$. ∎

6. Models

The concept of a model allows to reason semantically about term rewriting. The literature has focussed on algebraic models which are more difficult to get right — several notions of model in the literature are indeed flawed, see below. The reason they are tricky is that in order for semantic reasoning through an algebraic model A to be sound there needs to be a unique and continuous interpretation of $Ter^\infty(\Sigma)$ into A, and for that it does not suffice for A to be a Σ-algebra, since this does not account for infinite terms.

In [2] the issue of interpreting infinite terms was side-stepped: only interpretations of finite terms were provided a priori (thus rules were not allowed infinite terms in their right-hand sides). In fact, in the special case of equational models (rather than partially ordered ones), the construction in [2] specialises exactly to ordinary Σ-algebras satisfying a set of equations.

A class of algebraic models induces an equivalence relation on $Ter^\infty(\Sigma)$, i.e. t and u are equivalent iff they are the same in any model of that class. When looking at the equivalence derived from all models satisfying a set of equations we get the congruence closure of this set — when looking at models of $Ter(\Sigma)$. This is different for models of $Ter^\infty(\Sigma)$, because

the extra structure required in this class to guarantee for (unique) interpretations of infinite terms makes more equations true.

In [10] Zantema interpreted iTRSs in weakly monotone Σ-algebras with some extra structure. A special case are ordinary Σ-algebras since the ordering can be chosen to be equality. The extra structure required on a model A comprised there of: (i) a metric d_A; (ii) continuity of f_A for every function symbol f, w.r.t. the topology induced by the metric; (iii) the interpretation of the sequences $\mathrm{trunc}(t,n)$ in A converges for every infinite ground term t. Here, $\mathrm{trunc}(t,n)$ replaces all subterms of t at depth n with the fixed constant c.

As a notion of model this is flawed (see [4]) because the interpretation function from $\mathit{Ter}^\infty(\Sigma)$ to A derived from that is in general not continuous. Indeed, even the fix provided in [4] is insufficient because it is only expressed for ground terms, and any infinite term has non-ground terms in any neighbourhood; if $a \in A$ is arbitrary then the sequence $t_0 = a$, $t_{n+1} = f_A(t_n)$ could converge to a value different from the interpretation of f^∞, making the interpretation function non-continuous at f^∞.

This can be fixed by relating the notions of distance in A and $\mathit{Ter}^\infty(\Sigma)$: we can require the following condition for the distance function d_A on A:

$$d_A(f_A(a_1,\ldots,a_n), f_A(b_1,\ldots,b_n)) \leq \frac{1}{2} \cdot \max_{1\leq i\leq n} d_A(a_i,b_i)$$

This seemingly arbitrary condition derives as a special case from a more general condition we can set for metric models of other continuous term metrics [6].

Definition 6.1. Given a signature $\Sigma = (\mathcal{F}, \#)$ and continuous term metric m, a *metric model* is a Σ-algebra A, equipped with a metric $d_A : A \times A \to [0,1]$ such that (A, d_A) is a complete metric space and

$$d_A(f_A(a_1,\ldots,a_n), f_A(b_1,\ldots,b_n)) \leq f_m(d_A(a_1,b_1),\ldots,d_A(a_n,b_n))$$

for each $f \in \mathcal{F}, n = \#(f), a_i, b_i \in A$.

Here, f_m is the ultra-metric map associated with function symbol f in term metric m [6]. The condition on the order ensures that each f_A is continuous, but more importantly it leads to the following result:

Lemma 6.2. *Let A be a metric model (for Σ and m). Then the (unique) Σ-algebra homomorphism $[\![_]\!]_A : \mathit{Ter}(\Sigma) \to A$ is non-expansive.*

Proof. We need to show $d_A([\![t]\!]_A, [\![u]\!]_A) \leq d_m(t,u)$ for all finite terms t and u (note that $\mathit{Ter}(\Sigma)$ is the set of *finite* terms). The argument goes by induction on the size of t. If t is a constant then either $d_m(t,u) = 0$ which implies $t = u$ and thus $[\![t]\!]_A = [\![u]\!]_A$ and $d_A([\![t]\!]_A, [\![u]\!]_A) = 0$; or $d_m(t,u) = 1$ which is an upper bound for d_A.

Otherwise, $t = F(t_1,\ldots,t_n)$. Again, if $d_m(t,u) = 1$ the condition holds because 1 is an upper bound for d_A. If $d_m(t,u) \neq 1$ then u must be of the form $u = F(u_1,\ldots,u_n)$ and $d_m(t,u) = F_m(d_m(t_1,u_1),\ldots,d_m(t_n,u_n))$. Using the induction hypothesis on the t_i, monotonicity of F_m (all ultra-metric maps are monotonic) and the metric model order

property we get:

$$\begin{aligned} d_m(t,u) =& d_m(F(t_1,\ldots,t_n),F(u_1,\ldots,u_n)) \\ =& F_m(d_m(t_1,u_1),\ldots,d_m(t_n,u_n)) \\ \geq& F_m(d_A([\![t_1]\!]_A,[\![u_1]\!]_A),\ldots,d_A([\![t_n]\!]_A,[\![u_n]\!]_A)) \\ \geq& d_A(F_A([\![t_1]\!]_A,\ldots,[\![t_n]\!]_A),F_A([\![u_1]\!]_A,\ldots,[\![u_n]\!]_A)) \\ =& d_A([\![F(t_1,\ldots,t_n)]\!]_A,[\![F(u_1,\ldots,u_n)]\!]_A) \\ =& d_A([\![t]\!]_A,[\![u]\!]_A) \end{aligned}$$

■

Lemma 6.2 implies that $[\![_]\!]_A$ can be uniquely lifted to the respective metric completions (maintaining continuity), because non-expansive maps are uniformly continuous. Since A is already complete this lifts $[\![_]\!]_A$ to type $Ter^m(\Sigma) \to A$. Therefore we get an interpretation of infinite terms "for free".

Definition 6.3. The metric theory of a set of $Ter^m(\Sigma)$-equations E is the set of all pairs $(t,u) \in Ter^m(\Sigma) \times Ter^m(\Sigma)$ such that the equation $t = u$ holds in all metric models (w.r.t. signature Σ and term metric m) that satisfy the equations in E.

Theorem 6.4. *The metric theory of E w.r.t. Σ and m is strongly separating.*

Proof. We need to show that the theory is closed under limits of converging sequences. A sequence of pairs is converging in $Ter^m(\Sigma) \times Ter^m(\Sigma)$ iff the sequences of its first and second projections to $Ter^m(\Sigma)$ are. Suppose p is a sequence of pairs in the metric theory and that it is converging. Let the sequences l_n and r_n be their first and second projections, with limits l and r, respectively. Consider any metric model A of E. Using lemma 6.2 it is clear that the interpretation preserves convergence, hence $[\![l_n]\!]_A$ and $[\![r_n]\!]_A$ converge to $[\![l]\!]_A$ and $[\![r]\!]_A$. Since each pair $p_i = (l_i, r_i)$ is in the metric theory we have $[\![l_i]\!]_A = [\![r_i]\!]_A$ for all i. Therefore these sequences are identical in A and thus $[\![l]\!]_A = [\![r]\!]_A$. As this holds for any metric model A the pair (l,r) must be in the metric theory too. ■

In general, we cannot always construct an initial model for E (from its theory), but a sufficient condition is that $Ter^m(\Sigma)$ (restricted to ground terms) is compact.

Theorem 6.5. *Any set of $Ter^\infty(\Sigma)$-equations E has an initial model, if Σ is finite.*

Proof. The proof goes by constructing the model I; most of the argument works any metric m and signature Σ: we can quotient the ground terms of $Ter^m(\Sigma)$ by the metric theory of E and set a distance function d_I as the pointwise supremum of all distance functions of models satisfying E.

Since E is included in its metric theory I clearly satisfies E.

Checking the triangle inequality: $\forall A. d_A(t,u) \leq d_A(s,t) + d_A(s,u)$ implies $\forall A. d_A(t,u) \leq d_I(s,t) + d_I(s,u)$ and thus also $d_I(t,u) = \sup_A(d_A(t,u)) \leq d_I(s,t) + d_I(s,u)$.

Checking the zero-axiom: $d_I(t,u) = 0$ if $d_A(t,u) = 0$ for all A, i.e. if $t = u$ in all models. Hence $t = u$ in I too. If $t = u$ in I then $t = u$ in all A, hence $d_A(t,u) = 0$ in all A and so $d_I(t,u) = 0$.

Checking the metric model inequation:

$$\begin{aligned}
&d_I(f_I([\![a_1]\!]_I,\dots,[\![a_n]\!]_I), f_I([\![b_1]\!]_I,\dots,[\![b_n]\!]_I))\\
&\quad = d_I([\![f(a_1,\dots,a_n)]\!]_I, [\![f(b_1,\dots,b_n)]\!]_I)\\
&\quad = \sup_A(d_A([\![f(a_1,\dots,a_n)]\!]_A, [\![f(b_1,\dots,b_n)]\!]_A))\\
&\quad = \sup_A(d_A(f_A([\![a_1]\!]_A,\dots,[\![a_n]\!]_A), f_A([\![b_1]\!]_A,\dots,[\![b_n]\!]_A)))\\
&\quad \le \sup_A f_m(d_A([\![a_1]\!]_A,[\![b_1]\!]_A),\dots,d_A([\![a_n]\!]_A,[\![b_n]\!]_A))\\
&\quad \le f_m(\sup_A(d_A([\![a_1]\!]_A,[\![b_1]\!]_A),\dots,\sup_A(d_A([\![a_n]\!]_A,[\![b_n]\!]_A))))\\
&\quad = f_m(d_I([\![a_1]\!]_I,[\![b_1]\!]_I),\dots,d_I([\![a_n]\!]_I,[\![b_n]\!]_I))
\end{aligned}$$

We still need to show that the metric space (I, d_I) is complete, and for this we need theorem 5.18, and thus specialise the metric. The semantic interpretation $[\![_]\!]_I$ is continuous, and as its domain is compact and codomain T_2, it is also a closed map. Hence the limit of any Cauchy-sequence in (I, d_I) is in the image of that interpretation. ∎

7. Conclusion and Remark

We had a look at large number of alternative notions for transfinite reduction relations and transfinite equivalences. Of particular interest are the relations $\twoheadrightarrow_p$ and $\twoheadrightarrow_t$, as they are the ones most tightly coupled with transfinite equivalences, i.e. the separating and strongly separating equivalence closures of rewrite steps.

Notice that several of the examples show "undesirable" consequences in equational reasoning. This is due to the equational constraints implicit in $Ter^\infty(\Sigma)$, e.g. that all functions F have unique fixpoints F^∞. These problems can be addressed by a different choice of term metric that disallows certain infinite terms, because F can have multiple (or no) fixpoints in the absence of F^∞.

References

[1] Hendrik P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* North-Holland, 1984.

[2] Nachum Dershowitz and Stéphane Kaplan. Rewrite, Rewrite, Rewrite, Rewrite, Rewrite, ... In *Principles of Programming Languages*, pages 250–259. ACM, 1989.

[3] Nachum Dershowitz, Stéphane Kaplan, and David Plaisted. Rewrite, Rewrite, Rewrite, ... *Theoretical Computer Science*, 83(1):71–96, 1991.

[4] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Jan Willem Klop, and Roel Vrijer. Proving infinitary normalization. pages 64–82, 2009.

[5] Theordore W. Gamelin and Robert Everist Greene. *Introduction to Topology.* Dover, 1999.

[6] Stefan Kahrs. Infinitary rewriting: Meta-theory and convergence. *Acta Informatica*, 44(2):91–121, May 2007.

[7] Stefan Kahrs. Modularity of convergence in infinitary rewriting. In Ralf Treinen, editor, *Rewriting Techniques and Applications*, volume 5595 of *LNCS*, pages 179–193. Springer, 2009.

[8] Richard Kennaway, Jan Willem Klop, Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, 1995.

[9] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.

[10] Hans Zantema. Normalisation of infinite terms. In *RTA 2008*, volume 5117 of *LNCS*, pages 441–455, 2008.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 177-192
http://rewriting.loria.fr/rta/

COMPUTING RELATIVE NORMAL FORMS IN REGULAR TREE LANGUAGES

ALEXANDER KOLLER [1] AND STEFAN THATER [1]

[1] Saarland University
Saarbrücken, Germany
E-mail address: koller@mmci.uni-saatland.de
E-mail address: stth@coli.uni-saarland.de

ABSTRACT. We solve the problem of computing, out of a regular language L of trees and a rewriting system R, a regular tree automaton describing the set $L' \subseteq L$ of trees which cannot be R-rewritten into a tree in L. We call the elements of L' the relative normal forms of L. We apply our algorithm to the problem of computing weakest readings of sentences in computational linguistics, by approximating logical entailment with a rewriting system, and give the first efficient and practically useful algorithm for this problem. This problem has been open for 25 years in computational linguistics.

1. Introduction

One key task in computational linguistics is to represent the meaning of a natural language sentence using some formal representation *(reading)*, and to model inference on the level of natural language [1] as inference on the corresponding meaning representations. The classical approach [2] uses logical languages, such as first or higher order predicate logic, to represent sentence meanings. But when the sentence is *ambiguous*, it is often infeasible to explicitly enumerate all the different readings: The number of readings is worst-case exponential in the length of the sentence, and it is not uncommon for a sentence to have millions of readings if they contain a number of ambiguous constituents.

The standard technique to address this issue is *underspecification* [3, 4, 5, 6]: all readings of an ambiguous sentence are represented by a single, compact *underspecified representation (USR)*, such as a dominance graph [7]. Individual readings can be enumerated from an USR if necessary, but this step is postponed for as long as possible. This offers a partial solution to the problem of managing ambiguity. However, it is much less clear how to *disambiguate* a sentence, i.e. to determine the reading that the speaker actually intended in the given context.

In the absence of convincing disambiguation techniques, it has been proposed to work with the *weakest readings* of a sentence in practical applications [8]: If the readings are a set of formulas (say, of predicate logic), the weakest readings are those readings that are minimal with respect to logical entailment. From an application perspective, the weakest readings capture the “safe” information that is common to all possible readings; they are also linguistically interesting [9]. Because there are so many readings, it is infeasible to compute all readings and test all pairs for entailment using a theorem prover. However, although the problem has been open for over 25 years [9, 10], the best

Key words and phrases: normal forms, tree automata, incomplete inference, computational linguistics.

© A. Koller and S. Thater
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.177

known algorithm [11] is still quadratic in the number of readings and therefore too slow for practical use.

In this paper, we solve this problem for a sound but incomplete approximation of entailment by means of a rewrite system. Technically, we consider the problem of computing, from a regular tree language L of trees and a rewrite system R, a regular tree automaton representing the subset $L' \subseteq L$ consisting of all trees that cannot be R-rewritten into a tree in L. We call the elements of L' the *relative normal forms* of L with respect to R. To do this, we first represent the one-step rewriting relation of R in terms of a linear *context tree transducer*, which extends ordinary tree transducers by rewriting an entire context in each derivation step instead of a single symbol. We then compute the pre-image of L under this transducer, and intersect L with the complement of the pre-image. We show that an automaton accepting the pre-image can be computed in linear time if L is represented by a deterministic automaton. For a certain special case, which holds in our application, we show that we can even obtain a *deterministic* automaton for the pre-image in linear time. Altogether, we obtain an algorithm for computing weakest readings that is quadratic in the size of the tree automaton describing L, instead of quadratic in the size of L.

Despite the incompleteness, the approximation of entailment as rewriting is sufficient in our application: For one specific rewrite system, our algorithm computes a mean of 4.5 weakest readings per sentence in a text corpus, down from about three million readings that the sentences have on average. It takes 20 ms per sentence on average to do this. Thus, we see our algorithm as a practical solution to the problem of computing weakest readings in computational linguistics. On the other hand, our algorithm handles arbitrary linear rewriting systems and is therefore much more generally applicable. For instance, our earlier work on redundancy elimination [12], which was based on tree automata intersection as well, falls out as a special case; and we anticipate that further approximative inference techniques for natural language will be developed based on this paper in the future.

Plan of the paper. In Sect. 2, we define the problem and review dominance graphs and tree automata. We then define context tree transducers, use them to compute the pre-image of L under R, and analyze the complexity of the algorithm in Sect. 3. We go through an example from the application in Sect. 4, and conclude in Sect. 5.

2. Definitions

We start by reviewing some dominance graph theory and defining the problem we want to solve.

2.1. Dominance graphs

Semantic ambiguity, which is present when a natural-language sentence can have more than one possible meaning, is a serious problem in natural language processing with large-scale grammars. For instance, the mean number of possible meaning representations per sentence in the Rondane text corpus [13] is about $5 \cdot 10^9$. It is obviously impractical to enumerate all of these meaning representations. Instead, computational linguists typically use *underspecification* approaches, in which a single compact *description* of the possible meanings is computed instead of all possible semantic readings.

One formal approach to underspecification, which we use in this paper, is that of using *dominance graphs* [7]. Dominance graphs assume that the individual semantic representations of the sentence – say, formulas of predicate logic – are represented as trees, and then describe sets of trees

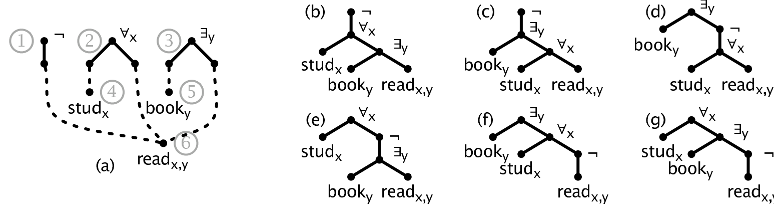


Figure 1: A dominance graph that represents the six readings of the sentence *every student did not read a book* (a) and its six configurations (b – g).

by specifying parent and ancestor relationships between nodes. They are equivalent to leaf-labeled normal dominance constraints [4].

We assume finite ranked signatures $\Sigma, \Sigma', \ldots$ of tree constructors f with arities $\mathsf{ar}(f)$. We define a *(finite constructor) tree t* over Σ to be a function $t : D \to \Sigma$, where D is a tree domain (i.e., a finite subset of $\mathbb{N}^*$ that is closed under prefix and left sibling), such that every node $u \in D$ has $\mathsf{ar}(t(u))$ many children. Alternatively, we can see each tree as a ground term over Σ. We write T_Σ for the set of trees over Σ.

Definition 2.1. A *(compact) labeled dominance graph* over a ranked signature Σ is a quadruple $G = (V, E \uplus D, L, <)$, where $(V, E \uplus D)$ is a directed graph, $L : V \rightsquigarrow \Sigma$ is a partial *(node) labeling function* and $< \subseteq V \times V$ a strict linear order on V, such that

(1) the graph (V, E) defines a collection of node disjoint trees of height 0 or 1 (we call the edges in E *tree edges*, the trees *fragments*, the roots of the fragments *roots* and all other nodes *holes*);
(2) if $(v, v') \in D$, then v is a hole and v' is a root in G (we call the edges in D *dominance edges*);
(3) the labeling function L assigns a node v a label with arity n iff v is a root with n outgoing tree edges (we write $f|_n$ to indicate that f has arity n);
(4) every hole has at least one outgoing dominance edge.

We write W_G for the roots and L_G for the labeling function of G, and we will say that v is a *hole of u* if $(u, v) \in E$. We will typically just say *dominance graph* for "compact labeled dominance graph".

Dominance graphs can be seen as descriptions of sets of trees, which can be obtained from the graph by "plugging" roots into holes so that dominance edges are realized as dominance. We call these trees the *configurations* of the graph. An example graph (for the sentence "every student did not read a book") and its six configurations are shown in Fig. 1, where we draw tree edges as solid lines and dominance edges as dotted lines, directed from top to bottom. The signature includes the symbols $\neg|_1$, $\forall_x|_2$, and $\mathsf{stud}_x|_0$; we read the trees over this signature as simplified formulas of predicate logic, taking $\forall_x(P, Q)$ to abbreviate $\forall x(P \to Q)$ and $\exists_x(P, Q)$ for $\exists x(P \wedge Q)$. Atomic formulas such as *stud*(x) are abbreviated by single function symbols such as *stud*$_x$ of arity 0. Now the six configurations, (b) – (g), are the six trees whose nodes are the (labeled) roots of the graph, such that all dominance edges in the graph are realized as reachability in the tree.

Formally, we define a *plugging* of a dominance graph $G = (V, E \uplus D, L, <)$ to be an injective partial function $p : V \rightsquigarrow V$ mapping each hole to a root. We can apply a plugging to a dominance graph to obtain a directed graph $p(G) = (V', E')$ such that $V' = W_G$ and $E' = \{(v, p(v')) \mid (v, v') \in E\}$. We call $p(G)$ an *unlabeled configuration* of G iff (i) $p(G)$ is a tree, and (ii) if $(v, v') \in D$, then $p(v)$ dominates v' in $p(G)$, i.e., there is a directed path from $p(v)$ to v' in the tree $p(G)$. By taking W_G as a ranked signature (with $\mathsf{ar}(u) = \mathsf{ar}(L_G(u))$) and ordering the children of each node according to $<$, we can read $p(G)$ as a finite constructor tree $t \in T_{W_G}$. An unlabeled configuration t can be mapped to

a finite constructor tree $L_G(t) \in T_\Sigma$ – called a *labeled configuration* – by labeling each node $u \in V'$ with $L_G(u)$. The configurations in Fig. 1 are all labeled. We say that a graph is *configurable* if it has a (labelled or unlabelled) configuration.

Throughout this paper, we restrict ourselves to *hypernormally connected* dominance graphs. We say that G is *hypernormally connected (hnc)* iff each pair of nodes is connected by a simple *hypernormal path*. A hypernormal path [7] in a dominance graph G is a path in the undirected version of G that does not use two dominance edges that are incident to the same hole. Hnc graphs have a number of desirable properties. For instance, the problem of deciding whether a dominance graph has a configuration is NP-complete in general, but polynomial if the graph is hnc. Furthermore, hnc dominance graphs can be translated into equivalent tree automata (see below). Note that virtually all dominance graphs that are used in linguistics applications are hnc [14].

2.2. Weakest readings

In order to perform inferences on the semantic representations for a sentence, it is desirable to identify the "correct" semantic representation from among all the (many) possibilities. Unfortunately, there are no satisfying models that would allow this. One alternative that has been proposed as a workaround is to compute the *weakest readings* – that is, the least informative semantic representations [8]. This idea exploits the fact that a set of predicate logic formulas is partially ordered with respect to logical entailment; the weakest readings are then the (configurations representing the) minimal elements of this order. In Fig. 1, (f) entails (g), (b) entails (c), and so on; (d) and (g) are incomparable, and indeed, (d) and (g) are the weakest readings of the dominance graph in Fig. 1a. The problem that motivates this paper is how to efficiently compute the weakest readings of a dominance graph.

A brute-force approach to this problem would be to compute all labeled configurations of a dominance graph, and then to run a theorem prover for each pair of configurations to establish the entailment order. Although this is clearly impractically slow when real-world sentences have an average of several billions of readings, the best known algorithm [11] is essentially just an optimization of this approach, and in particular is quadratic in the number of configurations.

Here we take a different approach. We will work with a sound but incomplete approximation of entailment using a rewriting system. Notice that the entailment between (f) and (g) can be explained by the fact that (f) can be rewritten into (g) by applying the rewrite rule

$$\exists_y(P, \forall_x(Q,R)) \to \forall_x(Q, \exists_y(P,R))^1 \tag{2.1}$$

if x does not occur in P. In positive contexts, the right-hand side of this rule is always entailed by the left-hand side; in this sense, the rule is sound. Now (g) can be recognized as a weakest reading because it cannot be rewritten into another tree which is also a configuration of the dominance graph.

In order to obtain a sound model of first-order entailment, we must make the rewriting system sensitive to the logical polarity of the subformula at which we apply a rewrite rule: If we were to apply (2.1) to the configuration (c), we would rewrite it into (b), which is logically *stronger* than (c). That is, we must restrict (2.1) such that it can only be used for subformulas that occur in a logically positive context.

[1] $\forall_x, \exists_y \in \Sigma$ are uninterpreted binary constructors. To ensure finiteness of the rewrite systems we use, we assume there is only a finite set of variables $x, y, \ldots$ in the language of semantic representations. P, Q, R are ordinary variables of the rewrite system.

More generally, we assume a finite alphabet Ann of *annotations*; we want to assign a single annotation to every node of a tree in T_Σ. We assign a *starting annotation* $a_0 \in \mathsf{Ann}$ to the root of each tree, and use an *annotator function* $\mathsf{ann} : \mathsf{Ann} \times \Sigma \times \mathbb{N} \to \mathsf{Ann}$ to compute the annotations for the other nodes: If some node u is annotated with a and has label $f \in \Sigma$, then we annotate the i-th child of u with $\mathsf{ann}(a, f, i)$ for all $1 \leq i \leq \mathsf{ar}(f)$. We then define an *annotated rewriting system R* as a finite set of pairs $a : r$, where $a \in \mathsf{Ann}$ and r is a rewrite rule over Σ. A tree $t \in \Sigma$ can be rewritten into a tree t', $t \to_R t'$, if there is a rule $a : r$ and a node u in t with annotation a such that t rewrites into t' by applying r at node u. We write $\to_R^*$ for the reflexive, transitive closure of $\to_R$.

Using this terminology, we can capture logical polarities by using $\mathsf{Ann} = \{+, -\}$, $a_0 = +$, and ann such that $\mathsf{ann}(+, \forall_x, 1) = -$, $\mathsf{ann}(+, \forall_x, 2) = +$, and so on. We can then rephrase (2.1) more precisely as follows:

$$+ : \exists_y(P, \forall_x(Q, R)) \to \forall_x(Q, \exists_y(P, R)) \tag{2.2}$$

This rule will still rewrite (f) into (g) because it is applied at the root (with annotation $+$), but it will not rewrite (c) into (b), because the redex has annotation $-$. We will extend (2.2) to a full rewrite system, which correctly characterizes (d) and (g) as the only two weakest readings, in Section 4.

Now observe that if we have an annotated rewriting system in which every rule makes the formula logically weaker, then all weakest readings will have the property that it is not possible to rewrite them into some other configuration. More generally, we will say that all weakest readings are in *relative normal form* with respect to the set of all configurations.

Definition 2.2. Let L be a set of trees over some signature Σ, and let R be an (annotated) rewrite system over Σ. We say that a tree $t \in L$ is in *relative normal form* with respect to R iff there is no tree $t' \in L$ such that $t \to_R t'$. We write $\mathrm{RNF}_R(L)$ for the relative normal forms in L with respect to R.

In the example, (d) and (g) are in relative normal form because they are in normal form, i.e. they cannot be rewritten at all. However, in general a tree may be in relative normal form without being in normal form, if all possible results of rewrites are not in L. For example, consider the set $L = \{f(f(g(h(a)))), f(g(f(h(a)))), f(f(h(g(a))))\}$ and a rewrite system R that consists of the single rule $f(g(x)) \to g(f(x))$. The tree $f(f(g(h(a))))$ rewrites to $f(g(f(h(a)))) \in L$, and is therefore not in relative normal form. However, while we could further rewrite this tree into $g(f(f(h(a))))$, the result is no longer in L, so $f(g(f(h(a))))$ is in relative normal form. The tree $f(f(h(g(a))))$ does not contain a redex in the first place, and is therefore also in relative normal form.

2.3. Dominance graphs as tree automata

The problem that we solve in this paper is to find an efficient algorithm for computing the relative normal forms in regular tree languages with respect to an annotated rewriting system. This solves the problem of computing the weakest readings of a dominance graph because the configuration sets of dominance graphs are regular tree languages, and it is known how to compute tree automata for accepting them. We will now recall some definitions regarding tree automata and transducers, and then sketch the translation of dominance graphs as tree automata.

Let Σ be a finite ranked signature as above, and let $\mathscr{X}$ be a finite set of (variable) symbols. We write $T_\Sigma(\mathscr{X})$ for $T_{\Sigma \cup \{a|_0 \mid a \in \mathscr{X}\}}$. If $\mathscr{X}_m$ is a set of m variables, we write $\mathrm{Con}^{(m)}(\Sigma)$ for the *contexts* with m holes, i.e. those trees in $T_\Sigma(\mathscr{X}_m)$ in which each element of $\mathscr{X}_m$ occurs exactly once. If $C \in \mathrm{Con}^{(m)}(\Sigma)$, then $C[t_1, \ldots, t_m] = C[t_1/x_1, \ldots, t_m/x_m]$, where $x_1, \ldots, x_m$ are the variables from left to right.

Definition 2.3. A *top-down tree transducer* from Σ to Δ is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, \delta)$, where Q is a finite set of states, Σ and Δ are ranked signatures, and $q_0 \in Q$ is the initial state. The rules in δ are of

$$q_{\{1,2,3,4,5,6\}}(\neg(x_1)) \to \neg(q_{\{2,3,4,5,6\}}(x_1))$$
$$q_{\{1,2,3,4,5,6\}}(\forall_x(x_1,x_2)) \to \forall_x(q_{\{4\}}(x_1), q_{\{1,3,5,6\}}(x_2))$$
$$q_{\{1,2,3,4,5,6\}}(\exists_y(x_1,x_2)) \to \exists_y(q_{\{5\}}(x_1), q_{\{1,2,4,6\}}(x_2))$$
$$q_{\{3,5,6\}}(\exists_y(x_1,x_2)) \to \exists_y(q_{\{5\}}(x_1), q_{\{6\}}(x_2))$$
$$q_{\{2,3,4,5,6\}}(\forall_x(x_1,x_2)) \to \forall_x(q_{\{4\}}(x_1), q_{\{3,5,6\}}(x_2))$$
$$q_{\{2,3,4,5,6\}}(\exists_y(x_1,x_2)) \to \exists_y(q_{\{5\}}(x_1), q_{\{2,4,6\}}(x_2))$$
$$q_{\{1,3,5,6\}}(\exists_y(x_1,x_2)) \to \exists_y(q_{\{5\}}(x_1), q_{\{1,6\}}(x_2))$$
$$q_{\{1,3,5,6\}}(\neg(x_1)) \to \neg(q_{\{3,5,6\}}(x_1))$$
$$q_{\{1,2,4,6\}}(\forall_x(x_1,x_2)) \to \forall_x(q_{\{4\}}(x_1), q_{\{1,6\}(x_2)})$$
$$q_{\{1,2,4,6\}}(\neg(x_1)) \to \neg(q_{\{2,4,6\}}(x_1))$$
$$q_{\{2,4,6\}}(\forall_x(x_1,x_2)) \to \forall_x(q_{\{4\}}(x_1), q_{\{6\}}(x_2))$$
$$q_{\{1,6\}}(\neg(x_1)) \to \neg(q_{\{6\}}(x_1))$$
$$q_{\{4\}}(\mathsf{stud}_\mathsf{x}) \to \mathsf{stud}_\mathsf{x}$$
$$q_{\{5\}}(\mathsf{book}_\mathsf{y}) \to \mathsf{book}_\mathsf{y}$$
$$q_{\{6\}}(\mathsf{read}_\mathsf{x,y}) \to \mathsf{read}_\mathsf{x,y}$$

Figure 2: A tree automaton accepting the labeled configurations of the dominance graph in Fig. 1(a).

the form $q(f(x_1,\dots,x_n)) \to C[q_1(x_{i_1}),\dots,q_m(x_{i_m})]$, where $f \in \Sigma$, $q,q_1,\dots,q_m \in Q$, $C \in \mathrm{Con}^{(m)}(\Delta)$, and $x_{i_k} \in \{x_1,\dots,x_n\}$ for all k.

If t is a tree in $T_{\Sigma\cup\Delta\cup Q}$, then we say that M derives t' in one step from t, $t \to_M t'$, if there are a context C', trees $t_1,\dots,t_n$ and a transition rule $q(f(x_1,\dots,x_n)) \to C[q_1(x_{i_1}),\dots,q_m(x_{i_m})]$ such that $t = C'[q(f(t_1,\dots,t_n))]$ and $t' = C'[C[q_1(t_{i_1}),\dots,q_m(t_{i_m})]]$. The *derivation relation* $\to^*$ of M is the reflexive, transitive closure of $\to$. The *translation relation* τ_M of M is

$$\tau_M = \{(t,t') \mid t \in T_\Sigma \text{ and } t' \in T_\Delta \text{ and } q_0(t) \to^* t'\}.$$

A tree transducer is called *linear* if no variable occurs twice, and *non-deleting* if every variable occurs at least once on the right-hand side of each rule. It is called *deterministic* if for every $q \in Q$ and $f \in \Sigma$, there is at most one rule whose left-hand side is $q(f(x_1,\dots,x_n))$.

A *top-down tree automaton* over Σ is a top-down transducer $A = (Q,\Sigma,\Sigma,q_0,\delta)$ such that every rule in δ is of the form $q(f(x_1,\dots,x_n)) \to f(q_1(x_1),\dots,q_n(x_n))$. We write $\mathscr{L}(A) = \{t \mid (t,t) \in \tau_A\}$ for the *language* accepted by A.

A *bottom-up tree automaton* is a 4-tuple $A = (Q,\Sigma,Q_F,\delta)$ in which $Q_F \subseteq Q$ is the set of *final states* and the transition rules in δ are of the form $f(q_1(x_1),\dots,q_n(x_n)) \to q(f(x_1,\dots,x_n))$. Derivations and languages are defined in analogy to the top-down case, see [15] for details. A bottom-up automaton is called deterministic if for every $f \in \Sigma$ and $q_1,\dots,q_n \in Q$ there is at most one rule whose left-hand side is $f(q_1(x_1),\dots,q_n(x_n))$.

For every top-down automaton A, there is a bottom-up automaton A' with $\mathscr{L}(A) = \mathscr{L}(A')$. For every bottom-up automaton A, there is a deterministic bottom-up automaton A' with $\mathscr{L}(A) = \mathscr{L}(A')$.

Now any hnc dominance graph G can be translated into a top-down tree automaton A_G that accepts the language of all labeled configurations, and into a deterministic top-down tree automaton A_G^u that accepts the language of all unlabeled configurations of G [12]. The states of these automata correspond to hnc subgraphs of G, and the transition rules encode decompositions of this subgraph into smaller subgraphs by removing a *free* root and its holes. A root u in a configurable graph G is called free iff G has a configuration whose root is u; it can be tested in linear time whether a root in a configurable hnc graph is free [16].[2]

The tree automaton we obtain for the labeled configurations of the graph in Fig. 1 (a) is shown in Fig. 2. The first rule states that we can choose $\neg$ as the root of a configuration, and obtain the subgraph $\{2,3,4,5,6\}$ by removing it. We can then choose $\forall_x$ as the root in this subgraph, splitting it into two weakly connected components, $\{4\}$ and $\{3,5,6\}$, and so on. This automaton

[2]The algorithm in [16] is defined in terms of *solved forms*. Our definition is equivalent for hnc dominance graphs.

accepts exactly the six labeled configurations of the original dominance graph. In practice, the tree automata computed for dominance graphs remain small; for instance, the graphs obtained for the sentences in the Rondane treebank [13] contain on average 14 roots and have $5 \cdot 10^9$ configurations, whereas the automata have 320 rules and can be computed in 20 ms on average [12].

3. Computing relative normal forms

We will now show how relative normal forms of regular tree languages with respect to a linear rewriting system can be computed. We will first sketch the basic idea and show an (inefficient) solution based on pre-images of regular tree languages under regular tree translations. We will then introduce *context tree transducers*, which allow us to use linear transducers and obtain an efficient algorithm.

Throughout, we limit ourselves to linear rewriting systems, which are sufficient for our application.

3.1. Relative normal forms as non-pre-images

As we defined above, a tree t in some set L of trees is in relative normal form iff there is no other tree t' in L into which t can be rewritten in one step. This can have two possible reasons: Either t is in normal form, i.e. there is no rewrite step that can be applied to it at all; or t can be rewritten, but no possible result of these rewrite steps is in L.

The key idea in this paper is to model the one-step rewriting relation of a rewriting system R with a top-down tree transducer M_R, such that $t \to_R t'$ iff $(t,t') \in \tau_R$. Given such a transducer, we can then determine the relative normal forms as those trees that cannot be rewritten with the transducer.

Lemma 3.1. *Let L be a set of trees, let R be a rewriting system, and let M be a transducer such that $t \to_R t'$ iff $(t,t') \in \tau_M$. Then*

$$\mathrm{RNF}_R(L) = L \cap \overline{\tau_M^{-1}(L)}.$$

For the case where L is a regular language of trees, the intersection can be computed efficiently using a construction on the tree automata. Complements of regular tree languages can also be computed on the automata themselves, although this requires computing the determinization of the tree automataton if it is nondeterministic, which may take exponential time. The question is how we can encode one-step rewriting in a tree transducer, and how we can compute the pre-image of L under τ_M.

For the first question, we can directly build a top-down transducer to encode the one-step rewriting relation. The transducer has two states, q and $\bar{q}$. The state q indicates that the transducer will apply a rewrite rule at some node below the current node; it is the start state. The state $\bar{q}$ indicates that the transducer will not apply any rewrite rule at the nodes below the current node; we require that all leaves of the tree are read in this state.

The transducer has two types of transitions. First, when the transducer is in state $\bar{q}$, it must copy the current symbol into the output tree; it may also choose to do this in state q:

$$\begin{array}{lcll} \bar{q}(f(x_1,\ldots,x_n)) & \to & f(\bar{q}(x_1),\ldots,\bar{q}(x_n)) & \text{for all } f \in \Sigma \\ q(f(x_1,\ldots,x_n)) & \to & f(\bar{q}(x_1),\ldots,q(x_i),\ldots,\bar{q}(x_n)) & \text{for some } 1 \le i \le n \text{ and all } f \in \Sigma \end{array} \quad (3.1)$$

In addition, in state q, M may choose to apply a rewrite rule and switch to state $\bar{q}$. Let's say we have a linear rewrite rule $f(g(x_1,x_2),x_3) \to g(x_1, f(x_2,x_3))$. We can represent an application of this rule with the following transition rules:

$$\begin{array}{lcl} q(f(x,y)) & \to & g(\bar{q}_{g,1}(x), f(\bar{q}_{g,2}(x), \bar{q}(y))) \\ \bar{q}_{g,1}(g(x,y)) & \to & \bar{q}(x) \\ \bar{q}_{g,2}(g(x,y)) & \to & \bar{q}(y), \end{array} \tag{3.2}$$

where crucially there are no other transitions for $\bar{q}_{g,1}$ and $\bar{q}_{g,2}$ than these, i.e. by using these states we simultaneously enforce the left-hand subtree below the f node to have root label g, and copy its two subtrees into the x_1 and x_2 positions of the rewriting rule.

In other words, it is possible to capture the one-step rewriting relation as the translation relation of a top-down tree transducer. But now consider how to compute $\tau_M^{-1}(L)$ for a regular tree language L. One possible idea is to consider a top-down tree transducer M_L such that $\tau_{M_L} = \{(t,t) \mid t \in L\}$; it is clear that such a transducer exists. We can then concatenate M and M_L; $\tau_M^{-1}(L)$ is the domain of $M \circ M_L$.

For deterministic tree transducers, it is known that the domain of a tree transducer is a regular tree language, and how to compute it ([17], Theorem 4.1; [18], p. 693). However, these algorithms are only applicable to *deterministic* transducers, and the transducers defined above are not deterministic (in state q, they may choose to either copy or rewrite). Furthermore, even if the transducers could be made deterministic, they compute regular tree automata of exponential size if the transducer is not linear. Indeed, the transducers shown above are not linear, because the first rule for the rewriting case duplicates x. This makes the use of these algorithms unattractive in our case.

3.2. Context tree transducers

However, although the above transducers can have non-linear rules, the extent of the non-linearity is limited: Every time a non-linear rule is applied, some other rules must be applied which will delete most of the copied trees, and retain only disjoint parts of them. This means that the machinery which the domain automaton construction uses to accommodate non-linear transducers is not necessary in our setting.

Consider the transition rules in (3.2). The only reason why we need to copy the g subtree twice is that we were not able to specify that the transducer should read the context $f(g(x_1,x_2),x_3)$ in the input tree directly. If we had a way to directly use this context on the left-hand side, each of x_1, x_2, and x_3 could appear only once on the right-hand side, and thus we could get away with a linear transducer. We will now extend the definition of top-down transducers in such a way that they can accept contexts on the left-hand side.

Definition 3.2. A *(top-down) context tree transducer* from Σ to Δ is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, \delta)$. δ is a finite set of transition rules of the form $q(C[x_1,\dots,x_n]) \to D[q_1(x_{i_1}),\dots,q_m(x_{i_m})]$, where $C \in \mathrm{Con}^{(n)}(\Sigma)$, $D \in \mathrm{Con}^{(m)}(\Delta)$, $q, q_1, \dots, q_m \in Q$, and $x_{i_k} \in \{x_1,\dots,x_n\}$ for all k.

If t is a tree in $T_{\Sigma\cup\Delta\cup Q}$, then we say that M derives t' in one step from t, $t \to t'$, if there is a context C', trees $t_1,\dots,t_n$, and a transition rule $q(C[x_1,\dots,x_n]) \to_M D[q_1(x_{i_1}),\dots,q_m(x_{i_m})]$ such that $t = C'[q(C[t_1,\dots,t_n])]$ and $t' = C'[D[q_1(t_{i_1}),\dots,q_m(t_{i_m})]]$. The *derivation relation* $\to_M^*$ is the reflexive, transitive closure of $\to_M$. The *translation relation* τ_M of M is

$$\tau_M = \{(t,t') \mid t \in T_\Sigma \text{ and } t' \in T_\Delta \text{ and } q_0(t) \to^* t'\}.$$

A context tree transducer is called *linear* if no variable occurs twice, and *non-deleting* if every variable occurs at least once on the right-hand side of each rule.

A *context tree automaton* is a context tree transducer with $\Sigma = \Delta$ and transition rules of the form $q(C[x_1,\ldots,x_n]) \to C[q_1(x_1),\ldots,q_n(x_n)]$. We take the *language* $\mathscr{L}(A)$ of a context tree automaton A to be the domain of τ_A.

Every top-down transducer is trivially also a top-down context transducer. Conversely, not every translation relation of a context transducer can be represented as the translation relation of an ordinary top-down transducer. For instance, the relation $\{(f(a,b),b)\}$ is the translation relation of a context transducer with rules $q(f(a,x)) \to q'(x)$ and $q'(b) \to b$. However, a top-down transducer must either output b when it reads the f or when it reads the b; either way, it must also accept an input tree $f(b,b)$. Every translation relation of a context transducer can also be computed by the "transformation language" of [19]. More specifically, context tree transducers are equivalent to extended left-hand side tree transducers (xTs) [20]. It is clear that an xT can encode a context transducer by specifying all constructors in the context in its tree pattern. Furthermore, an xT can be simulated in a context transducer by having a transition rule for every context C that satisfies the tree pattern on the left-hand side of each xT rule.

On the other hand, context *automata* are equivalent to ordinary tree automata. This can be shown as for regular tree grammars.

3.3. Rewriting with context tree transducers

Given a linear rewriting system R, it is now straightforward to produce a context tree transducer M_R whose translation relation is the one-step rewriting relation of R. First, M_R uses the rules in (3.1) to be able to copy parts of the input tree to the output unchanged. Second, for each rewrite rule $C[x_1,\ldots,x_n] \to C'[x_{i_1},\ldots,x_{i_n}]$, M_R contains a transition rule as follows:

$$q(C[x_1,\ldots,x_n]) \to C'[\bar{q}(x_{i_1}),\ldots,\bar{q}(x_{i_n})]. \tag{3.3}$$

Unlike the transducer in Section 3.1, this transducer is now linear. We will exploit this in Section 3.4 to obtain a more efficient algorithm for computing a pre-image.

But before we do this, let us extend the construction of context tree transducers for rewriting systems to annotated rewriting systems. Let's say we have an annotation alphabet Ann, an annotator function ann, and a linear annotated rewriting system R over Σ and Ann. We can obtain a context tree transducer M_R for the one-step rewriting relation of R by keeping track of the current annotation of nodes in the input tree in the state. In particular, we split the state q into states $q^{a_1},\ldots,q^{a_n}$ where $\mathsf{Ann} = \{a_1,\ldots,a_n\}$. We retain a single state $\bar{q}$, as no further rewriting can take place in this state, and the annotation is therefore irrelevant.

The initial state of M_R is q^{a_0}, where a_0 is the starting annotation. We then have the following versions of the transition rules in (3.1); these rules copy symbols to the output tree and keep track of the current annotation.

$$\begin{array}{lcll} \bar{q}(f(x_1,\ldots,x_n)) & \to & f(\bar{q}(x_1),\ldots,\bar{q}(x_n)) & \text{for all } f \in \Sigma \\ q^a(f(x_1,\ldots,x_n)) & \to & f(\bar{q}(x_1),\ldots,q^{\mathsf{ann}(a,f,i)}(x_i),\ldots,\bar{q}(x_n)) & \text{for some } 1 \le i \le n \text{ and all } f \in \Sigma \end{array} \tag{3.4}$$

In addition, M_R contains the following version of the transition rules in (3.3). It applies the rewriting rule $a : C[x_1,\ldots,x_n] \to C'[x_{i_1},\ldots,x_{i_n}]$ if the transducer is at a node in the input tree which is annotated with a.

$$q^a(C[x_1,\ldots,x_n]) \to C'[\bar{q}(x_{i_1}),\ldots,\bar{q}(x_{i_n})]. \tag{3.5}$$

Lemma 3.3. *Let t,t' be trees, and let R be a linear annotated rewrite system. Then $t \to_R t'$ iff $(t,t') \in \tau_{M_R}$.*

3.4. Pre-images under linear context tree translations

Finally, we show how to compute the pre-image of a regular tree language under the translation relation of a linear context tree transducer.

Proposition 3.4. *Let L be a regular tree language, and let M be a linear context tree transducer. Then $\tau_M^{-1}(L)$ is a regular tree language.*

Proof. Let $M = (P, \Sigma, \Delta, p_0, \delta)$, and let $B = (Q, \Delta, q_0, \gamma)$ be a top-down tree automaton with $\mathscr{L}(B) = L$. We construct a context tree automaton $A = (P \times Q, \Sigma, \Sigma, \langle p_0, q_0 \rangle, \eta)$ with $\mathscr{L}(A) = \tau_M^{-1}(L)$.

Let $p(C[x_1, \ldots, x_n]) \to D[p_1(x_{i_1}), \ldots, p_n(x_{i_n})]$ be in δ. Furthermore, let $q(D[x_1, \ldots, x_n]) \to_B^* D[q_1(x_1), \ldots, q_n(x_n)]$, where we extend $\to_B$ to a binary relation on $T_{Q \cup \Delta \cup \{x_1, \ldots, x_n\}}$ by using $x_i \to_B x_i$ for all i. Then we let A contain the transition rule

$$\langle p, q \rangle (C[x_1, \ldots, x_n]) \to C[\langle p_{k_1}, q_{k_1} \rangle (x_1), \ldots, \langle p_{k_n}, q_{k_n} \rangle (x_n)],$$

where $k_{i_j} = j$ for all j.

Intuitively, A should read a context C if M can translate this into a context which B accepts. We keep track of the states in which M and B are during this process in A's state. If A is in a state $\langle p, q \rangle$, M must translate C from state p; it will output a context D, which B must accept from state q. During its run, M assigns states $p_1, \ldots, p_n$ to the holes of C; similarly, B assigns states $q_1, \ldots, q_n$ to the holes of D. Because M is linear, the holes of C and D correspond to each other bijectively, and we can build the new states $\langle p_i, q_i \rangle$ in which A must then read the subtrees below the holes of C. ∎

3.5. Complexity

In the worst case, the algorithm in the proof of Prop. 3.4 constructs a tree automaton with at most $|P| \cdot |Q|$ states and at most $|\delta| \cdot |Q| \cdot m$ transition rules, where m is the maximum number of state tuples $(q_1, \ldots, q_n)$ which B can assign to the holes of any context D on the right-hand side of a rule in M. If B is deterministic, we have $m = 1$, i.e. we can construct the pre-image automaton in time $O(|B| \cdot |M|) = O(|B| \cdot |R|)$. This means that by exploiting the linearity, we avoid the exponential blow-up in the automaton size from the domain automaton construction in [17]. If B is nondeterministic, m may be exponential in the size of M, where the exponent is the maximum number of variables on the right-hand side of a transition rule.

One further complication is that the construction in Lemma 3.1 requires us to compute the complement of the pre-image automaton A. Even if B is deterministic, A may not be, and so the automaton for $\overline{\mathscr{L}(A)}$ may be exponentially larger because A must be determinized. However, the rewriting systems and tree automata that we use in our application have certain properties that make the deterministic pre-image automata small as well. We first define these special properties, and then prove the complexity result.

Definition 3.5. The *left-hand size* of a rewriting system R over Σ is the maximum number of constructors from Σ that is used on the left-hand side of a rule in R.

We call a top-down tree automaton *both way deterministic* if it is deterministic and the bottom-up tree automaton that is obtained by reversing all transition rules is also deterministic.

Now we can prove the key complexity result for our application.

Proposition 3.6. *Let B be a both way deterministic top-down tree automaton, and let R be a linear rewriting system of left-hand size at most 2. Then it is possible to compute a deterministic bottom-up tree automaton A such that $\mathscr{L}(A) = \tau_{M_R}^{-1}(\mathscr{L}(B))$ in time $O(|B| \cdot |R|)$.*

Proof. First, we build the nondeterministic top-down context automaton N with $\mathscr{L}(N) = \tau_{M_R}^{-1}(\mathscr{L}(B))$ as in the proof of Prop. 3.4. This automaton has three types of rules, of the following forms:

(1) $\langle p^a, q\rangle(f(x_1,\dots,x_n)) \to f(\langle \bar{p}, q_1\rangle(x_1),\dots,\langle p^{a'}, q_i\rangle(x_i),\dots,\langle \bar{p}, q_n\rangle(x_n))$
(2) $\langle \bar{p}, q\rangle(f(x_1,\dots,x_n)) \to f(\langle \bar{p}, q_1\rangle(x_1),\dots,\langle \bar{p}, q_n\rangle(x_n))$
(3) $\langle p^a, q\rangle(C[x_1,\dots,x_n]) \to C[\langle \bar{p}, q_1\rangle(x_1),\dots,\langle \bar{p}, q_n\rangle(x_n)]$

Rules of types 1 and 2 encode decisions of the transducer to copy symbols, whereas rules of type 3 encode a decision to rewrite C into some other context.

In a second step, we build an ordinary nondeterministic bottom-up tree automaton N' such that $\mathscr{L}(N') = \mathscr{L}(N)$. This can be done by breaking the transition rules for contexts with more than one constructor up into ordinary transition rules that read single constructors, and reversing the direction of all arrows. We can do this for the three rule types as follows:

(1) $f(\langle \bar{p}, q_1\rangle(x_1),\dots,\langle p^{a'}, q_i\rangle(x_i),\dots,\langle \bar{p}, q_n\rangle(x_n)) \to \langle p^a, q\rangle(f(x_1,\dots,x_n))$
(2) $f(\langle \bar{p}, q_1\rangle(x_1),\dots,\langle \bar{p}, q_n\rangle(x_n)) \to \langle \bar{p}, q\rangle(f(x_1,\dots,x_n))$
(3) If C is of the form $f(x, g(y,z))$ and there is a type 2 rule in N' of the form $g(\langle \bar{p}, q_2\rangle(x_2), \langle \bar{p}, q_3\rangle(x_3)) \to \langle \bar{p}, q'\rangle(g(x_2,x_3))$, then the rule $\langle p^a, q\rangle(f(x_1, g(x_2,x_3))) \to f(\langle \bar{p}, q_1\rangle(x_1), g(\langle \bar{p}, q_2\rangle(x_2), \langle \bar{p}, q_3\rangle(x_3)))$ gets broken up into the following two rules:

$$g(\langle \bar{p}, q_2\rangle(x_2), \langle \bar{p}, q_3\rangle(x_3)) \to \langle p^g, q'\rangle(g(x_2,x_3))$$
$$f(\langle \bar{p}, q_1\rangle(x_1), \langle p^g, q'\rangle(y)) \to \langle p^a, q\rangle(f(x_1,y)).$$

The form $f(g(x,y),z)$ is analogous. If C is of the form $f(x,y)$, then we simply reverse the rule into $f(\langle \bar{p}, q_1\rangle(x_1), \langle \bar{p}, q_2\rangle(x_2)) \to \langle p^a, q\rangle(f(x_1,x_2))$.

Finally, we determinize N' into a deterministic bottom-up automaton A such that $\mathscr{L}(A) = \mathscr{L}(N')$ and A does not contain states that are not reachable in a bottom-up run of the automaton. According to the standard construction, we have $Q_A \subseteq \mathscr{P}(Q_{N'})$; but which of these states are actually reachable? First, for any $\mathfrak{q} \in Q_A$, if $\langle p, q\rangle \in \mathfrak{q}$ and $\langle p', q'\rangle \in \mathfrak{q}$, then $q = q'$: Because B, if read as a bottom-up transducer, is deterministic, the q on the right-hand sides of type 1 and 2 rules in N' is uniquely determined by f and the $q_1,\dots,q_n$, and the type 3 rules are constructed to maintain this invariant too. Furthermore, we know that for every $\mathfrak{q} \in Q_A$, there is a $q \in Q_B$ such that $\langle \bar{p}, q\rangle \in \mathfrak{q}$, by induction using the type 2 rules. We also know that there is at most one p^f for $f \in \Sigma$ such that there is a q with $\langle p^f, q\rangle \in \mathfrak{q}$: namely the one for the most recent f that was read with a type 3 rule. Finally, $\mathfrak{q}$ may contain an arbitrary number of pairs of the form $\langle p^a, q\rangle$ for annotations a.

This means that $|Q_A| = O(|Q_B| \cdot |\Sigma| \cdot 2^{|\mathsf{Ann}|})$ where Ann is the annotation alphabet, i.e. the size of A's state alphabet is linear in that of B. In addition, A has at most as many transition rules as N'. If B has k rules and m is the maximum arity of symbols in Σ, this amounts to $k \cdot m$ rules of type 1, k rules of type 2, and at most $2 \cdot |R| \cdot |Q_B|$ rules of type 3. That is, the size of A's rule set is linear in k. Because both N and N' can be computed from B in linear time (as we argued above), this means that we compute A in linear time. ∎

In our application to scope underspecification, the tree automata A_G^u for the unlabeled configurations of a hnc dominance graph (e.g., the unlabeled version of the automaton in Fig. 2) are both way deterministic, and the rewriting rules we use (such as (2.2)) only permute two adjacent constructors, i.e. they are all of left-hand size two. In other words, for any A_G^u we can compute a deterministic automaton for the pre-image language in linear time. It is then straightforward to compute the complement automaton $\bar{A}$ and intersect it with A_G^u, obtaining an automaton A_W as the end result; this last step can take time $O(|A_G^u| \cdot |\bar{A}|) = O(|A_G^u|^2)$ in the worst case. Altogether we obtain an algorithm for

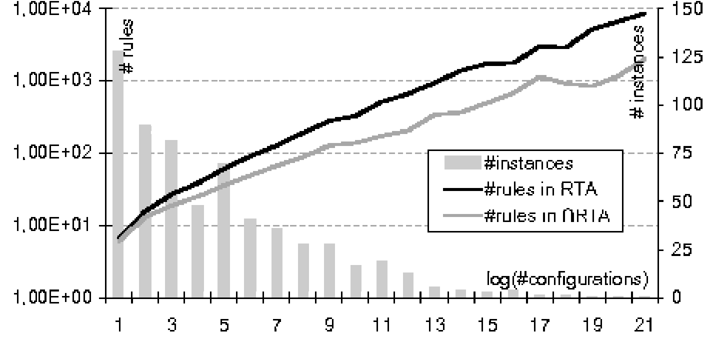


Figure 3: Sizes of automata in the Rondane treebank.

computing weakest readings that is quadratic in $|A^u_G|$, which is a huge improvement over the best previous algorithm, which was quadratic in $|\mathscr{L}(A^u_G)|$.

We have implemented a version of this algorithm which is further optimized to exploit certain properties of dominance graphs, and evaluated it empirically [21]. We ran the algorithm on the tree automata obtained from the dominance graphs for a subset of 623 sentences in the Rondane corpus [13]. Each of these dominance graphs describes a set of formulas of a variant of higher-order predicate logic, for which we chose suitable rewriting rules for approximating entailment. We find that the mean number of configurations represented by the automata drops from about three million for the original automata to 4.5 for the resulting automata A_W, and we reduce 67% of the sentences to a single weakest reading. The entire computation can be performed in about 20 ms per sentence on average. This means that although modeling weakest readings in terms of a rewriting system is incomplete with respect to true logical entailment, the approximation and our algorithm are highly useful on practical data.

It is interesting to observe that although the intersection construction can potentially make A_W larger than A^u_G, this does not actually happen in practice. This is shown in Fig. 3: The black line plots the sizes of the A^u_G, whereas the grey line plots the sizes of the A_W. The horizontal axis represents groups of sentences, where each group i contains all sentences with $\lceil e^{i-1} \rceil$ to $\lfloor e^i \rfloor$ readings; the vertical axis plots the mean number of transition rules in the automata (note the logarithmic scale). The grey bars indicate the number of sentences in each group. As the figure shows, the A_W tend to be much smaller than the original automata for each group. Averaged over all automata obtained from the subcorpus, the original automata have 180 transitions, whereas the result automata have 68; 87% of the automata are smaller after the intersection. It remains an open question to explain why the intersection decreases the automaton size so consistently.

4. An example

We finish by demonstrating our algorithm on the initial problem of computing the weakest readings of the dominance graph in Fig. 1(a). We assume the following annotated rewriting system, in which (2.2) is repeated as (4.1):

$$+ : \exists_y(P, \forall_x(Q,R)) \to \forall_x(Q, \exists_y(P,R)) \tag{4.1}$$

$$- : \forall_x(P, \exists_y(Q,R)) \to \exists_y(Q, \forall_x(P,R)) \tag{4.2}$$

$$+ : \neg(\exists_y(P,Q)) \to \exists_y(P, \neg(Q)) \tag{4.3}$$

$$+ : \forall_x(P, \neg(Q)) \to \neg(\forall_x(P,Q)) \tag{4.4}$$

This rewrite system translates into a top-down context tree transducer M_R with the following transition rules; we show the type 1 and 2 rules only for $\exists_y$.

$$p^+(\exists_y(x_1,\forall_x(x_2,x_3))) \to \forall_x(\bar{p}(x_2),\exists_y(\bar{p}(x_1),\bar{p}(x_3)))$$
$$p^+(\forall_x(x_1,\exists_y(x_2,x_3))) \to \exists_y(\bar{p}(x_2),\forall_x(\bar{p}(x_1),\bar{p}(x_3)))$$
$$p^+(\forall_x(x_1,\neg(x_2))) \to \neg(\forall_x(\bar{p}(x_1),\bar{p}(x_2)))$$
$$p^+(\neg(\exists_y(x_1,x_2))) \to \exists_y(\bar{p}(x_1),\neg(\bar{p}(x_2)))$$
$$p^+(\exists_y(x_1,x_2)) \to \exists_y(\bar{p}(x_1),p^+(x_2)) \qquad p^+(\exists_y(x_1,x_2)) \to \exists_y(p^+(x_1),\bar{p}(x_2))$$
$$p^-(\exists_y(x_1,x_2)) \to \exists_y(\bar{p}(x_1),p^-(x_2)) \qquad p^-(\exists_y(x_1,x_2)) \to \exists_y(p^-(x_1),\bar{p}(x_2))$$
$$\bar{p}(\exists_y(x_1,x_2)) \to \exists_y(\bar{p}(x_1),\bar{p}(x_2)) \qquad \ldots$$
$$\bar{p}(\mathsf{stud}_x) \to \mathsf{stud}_x \qquad \bar{p}(\mathsf{book}_y) \to \mathsf{book}_y \qquad \bar{p}(\mathsf{read}_{x,y}) \to \mathsf{read}_{x,y}$$

We can now consider an automaton A_G representing the configurations of a dominance graph, as in Fig. 2, and compute a nondeterministic top-down context automaton N with $L(N) = \tau_{M_R}^{-1}(L(A_G))$, as in the proof of Prop. 3.4. In the example, N looks as follows. Note that we are only showing transitions between productive states, and we abbreviate the state $\langle q_{G'}, p^a\rangle$ as $q^a_{G'}$. Note also that the construction in Prop. 3.4 would usually be executed on the unlabeled version A^u_G of A_G, but we show the labeled version here because A_G is already both ways deterministic in this example, and easier to read.

$$q^+_{1,\ldots,6}(\exists_y(x_1,\forall_x(x_2,x_3))) \to \exists_y(\bar{q}_5(x_1),\forall_x(\bar{q}_4(x_2),\bar{q}_{1,6}(x_3)))$$
$$q^+_{1,\ldots,6}(\neg(\exists_y(x_1,x_2))) \to \neg(\exists_y(\bar{q}_5(x_1),\bar{q}_{2,4,6}(x_2))) \qquad q^+_{1,\ldots,6}(\forall_x(x_1,x_2)) \to \forall_x(\bar{q}_4(x_1),q^+_{1,3,5,6}(x_2))$$
$$q^+_{1,\ldots,6}(\neg(x_1)) \to \neg(q^-_{2,\ldots,6}(x_1)) \qquad \bar{q}_{1,\ldots,6}(\exists_y(x_1,x_2))) \to \exists_y(\bar{q}_5(x_1),\bar{q}_{1,2,4,6}(x_2))$$
$$\bar{q}_{1,\ldots,6}(\forall_x(x_1,x_2)) \to \forall_x(\bar{q}_4(x_1),\bar{q}_{1,3,5,6}(x_2)) \qquad \bar{q}_{1,\ldots,6}(\neg(x_1)) \to \neg(\bar{q}_{2,\ldots,6}(x_1))$$
$$q^-_{2,\ldots,6}(\forall_x(x_1,\exists_y(x_2,x_3))) \to \forall_x(\bar{q}_4(x_1),\exists_y(\bar{q}_5(x_2),\bar{q}_6(x_3)))$$
$$\bar{q}_{2,\ldots,6}(\exists_y(x_1,x_2))) \to \exists_y(\bar{q}_5(x_1),\bar{q}_{2,4,6}(x_2)) \qquad \bar{q}_{2,\ldots,6}(\forall_x(x_1,x_2)) \to \forall_x(\bar{q}_4(x_1),\bar{q}_{3,5,6}(x_2))$$
$$q^+_{1,3,5,6}(\neg(\exists_y(x_1,x_2))) \to \neg(\exists_y(\bar{q}_5(x_1),\bar{q}_6(x_2)))$$
$$\bar{q}_{1,3,5,6}(\exists_y(x_1,x_2))) \to \exists_y(\bar{q}_5(x_1),\bar{q}_{1,6}(x_2)) \qquad \bar{q}_{1,3,5,6}(\neg(x_1)) \to \neg(\bar{q}_{3,5,6}(x_1))$$
$$\bar{q}_{1,2,4,6}(\forall_x(x_1,x_2)) \to \forall_x(\bar{q}_4(x_1),\bar{q}_{1,6}(x_2)) \qquad \bar{q}_{1,2,4,6}(\neg(x_1)) \to \neg(\bar{q}_{2,4,6}(x_1))$$
$$\bar{q}_{2,4,6}(\forall_x(x_1,x_2)) \to \forall_x(\bar{q}_4(x_1),\bar{q}_6(x_2)) \qquad \bar{q}_{3,5,6}(\exists_y(x_1,x_2))) \to \exists_y(\bar{q}_5(x_1),\bar{q}_6(x_2))$$
$$\bar{q}_{1,6}(\neg(x_1)) \to \neg(\bar{q}_6(x_1)) \qquad \bar{q}_4(\mathsf{stud}_x) \to \mathsf{stud}_x \qquad \bar{q}_5(\mathsf{book}_y) \to \mathsf{book}_y \qquad \bar{q}_6(\mathsf{read}_{x,y}) \to \mathsf{read}_{x,y}$$

Finally, we compute a deterministic ordinary bottom-up automaton A with $L(A) = L(N)$ as in the proof of Prop. 3.6. This involves breaking up rules whose left-hand sides are nontrivial contexts up into ordinary rules, reorienting the transitions into bottom-up rules, and determinizing the resulting automaton. The states of the determinized automaton are sets that contain states of N and states q^f_G that were introduced when breaking up the context rules; we suppress the set brackets for singleton sets below. Notice that as we claimed in the proof of Prop. 3.6, if any two of $\bar{q}_{G'}$ and $q^+_{G''}$ or $q^-_{G''}$ are in the same state set, then $G' = G''$; and there is at most one state $q^f_{G'}$ for any $f \in \Sigma$ in each such state set. A looks as follows:

$$\exists_y(\bar{q}_5(x_1), \{q^{\forall x}_{1,2,4,6}, \bar{q}_{1,2,4,6}\}(x_2)) \to \{q^{+}_{1,\ldots,6}, \bar{q}_{1,\ldots,6}\}(\exists_y(x_1,x_2))$$
$$\neg(\{q^{\exists y}_{2,\ldots,6}, \bar{q}_{2,\ldots,6}\}(x_1)) \to \{q^{+}_{1,\ldots,6}, \bar{q}_{1,\ldots,6}\}(\neg(x_1))$$
$$\forall_x(\bar{q}_4(x_1), \{q^{+}_{1,3,5,6}, \bar{q}_{1,3,5,6}\}(x_2)) \to \{q^{+}_{1,\ldots,6}, \bar{q}_{1,\ldots,6}\}(\forall_x(x_1,x_2))$$
$$\neg(\{q^{-}_{2,\ldots,6}, \bar{q}_{2,\ldots,6}\}(x_1)) \to \{q^{+}_{1,\ldots,6}, \bar{q}_{1,\ldots,6}\}(\neg(x_1))$$
$$\exists_y(\bar{q}_5(x_1), \bar{q}_{2,4,6}(x_2))) \to \{q^{\exists y}_{2,\ldots,6}, \bar{q}_{2,\ldots,6}\}(\exists_y(x_1,x_2))$$
$$\forall_x(\bar{q}_4(x_1), \{q^{\exists y}_{3,5,6}, \bar{q}_{3,5,6}\}(x_2)) \to \{q^{-}_{2,\ldots,6}, \bar{q}_{2,\ldots,6}\}(\forall_x(x_1,x_2))$$
$$\forall_x(\bar{q}_4(x_1), \bar{q}_{1,6}(x_2))) \to \{q^{\forall x}_{1,2,4,6}, \bar{q}_{1,2,4,6}\}(\forall_x(x_1,x_2))$$
$$\neg(\bar{q}_{2,4,6}(x_1)) \to \bar{q}_{1,2,4,6}(\neg(x_1))$$
$$\neg(\{q^{\exists y}_{3,5,6}, \bar{q}_{3,5,6}\}(x_1)) \to \{q^{+}_{1,3,5,6}, \bar{q}_{1,3,5,6}\}(\neg(x_1))$$
$$\forall_x(\bar{q}_4(x_1), \bar{q}_6(x_2)) \to \bar{q}_{2,4,6}(\forall_x(x_1,x_2))$$
$$\exists_y(\bar{q}_5(x_1), \bar{q}_6(x_2))) \to \{q^{\exists y}_{3,5,6}, \bar{q}_{3,5,6}\}(\exists_y(x_1,x_2))$$
$$\neg(\bar{q}_6(x_1)) \to \bar{q}_{1,6}(\neg(x_1)) \qquad \mathsf{stud}_x \to \bar{q}_4(\mathsf{stud}_x) \qquad \mathsf{book}_y \to \bar{q}_5(\mathsf{book}_y) \qquad \mathsf{read}_{x,y} \to \bar{q}_6(\mathsf{read}_{x,y})$$

A is a deterministic automaton which accepts four trees, namely the configurations (b), (c), (e), and (f) in Fig. 1. Therefore we can obtain an automaton which accepts exactly the weakest readings of the graph in Fig. 1 – i.e., (d) and (g) – by intersecting A_G and the complement automaton $\bar{A}$.

5. Conclusion

In this paper, we have presented an algorithm for computing those members of a regular tree language L that are in relative normal form with respect to an annotated rewriting system R. We have shown how to compute these elements by computing the pre-image of the tree language under a transducer encoding the one-step rewriting relation, and then intersecting the language with the complement of this pre-image. By defining *context* tree transducers, we were able to compute the pre-image in linear time if L is given in terms of a deterministic automaton; for the special case where R has left-hand sides of size at most two, we could show that even the deterministic automaton for the pre-image is linear in size. This restriction holds in our application to computational linguistics, where our results provide an approximate, but practically useful solution to the problem of computing weakest readings.

From the perspective of our application, our results open up a whole new class of rewriting-based inferences on natural-language meaning representations which can now be processed efficiently. We will explore such inferences in the future. One line of research that seems particularly intriguing is to deal with cases where multiple trees that are equivalent with respect to the underlying rewrite system are left over in the language of the final tree automaton. Such cases can happen when the rewrite system is not confluent. It would be interesting to investigate the practical impact of augmenting the permutation system, e.g. with the Knuth-Bendix completion procedure. This trades off a reduction in the number of relative normal forms (due to improved confluence) against an increase in the size of the rewrite system.

Acknowledgments. We would like to thank the reviewers and particularly Joachim Niehren for their extremely helpful comments, which influenced this paper substantially and improved it a lot.

References

[1] Dagan, I., Glickman, O., Magnini, B.: The PASCAL recognising textual entailment challenge. In Quiñonero-Candela, J., Dagan, I., Magnini, B., d'Alché Buc, F., eds.: Machine Learning Challenges. Volume 3944 of Lecture Notes in Computer Science. Springer (2006) 177–190

[2] Montague, R.: The proper treatment of quantification in ordinary English. In Thomason, R., ed.: Formal Philosophy. Selected Papers of Richard Montague. Yale University Press, New Haven (1974)

[3] van Deemter, K., Peters, S.: Semantic Ambiguity and Underspecification. CSLI (1996)

[4] Egg, M., Koller, A., Niehren, J.: The Constraint Language for Lambda Structures. Logic, Language, and Information **10** (2001) 457–485

[5] Copestake, A., Flickinger, D., Pollard, C., Sag, I.: Minimal recursion semantics: An introduction. Journal of Language and Computation (2005)

[6] Blackburn, P., Bos, J.: Representation and Inference for Natural Language. A First Course in Computational Semantics. CSLI Publications (2005)

[7] Althaus, E., Duchier, D., Koller, A., Mehlhorn, K., Niehren, J., Thiel, S.: An efficient graph algorithm for dominance constraints. Journal of Algorithms **48** (2003) 194–219

[8] Bos, J.: Let's not argue about semantics. In: Proceedings of LREC. (2008) 2835–2840

[9] Kempson, R., Cormack, A.: Ambiguity and quantification. Linguistics and Philosophy **4** (1981) 259–309

[10] Hobbs, J.: An improper treatment of quantification in ordinary English. In: Proceedings of the 21st ACL. (1983)

[11] Gabsdil, M., Striegnitz, K.: Classifying scope ambiguities. In: Proceedings of the First Intl. Workshop on Inference in Computational Semantics. (1999)

[12] Koller, A., Regneri, M., Thater, S.: Regular tree grammars as a formalism for scope underspecification. In: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-08: HLT), Columbus, Ohio (2008)

[13] Oepen, S., Toutanova, K., Shieber, S., Manning, C., Flickinger, D., Brants, T.: The LinGO Redwoods treebank: Motivation and preliminary applications. In: Proceedings of the 19th International Conference on Computational Linguistics (COLING'02). (2002) 1253–1257

[14] Fuchss, R., Koller, A., Niehren, J., Thater, S.: Minimal recursion semantics as dominance constraints: Translation, evaluation, and analysis. In: Proc. of ACL, Barcelona (2004)

[15] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata` (2007)

[16] Bodirsky, M., Duchier, D., Niehren, J., Miele, S.: An efficient algorithm for weakly normal dominance constraints. In: ACM-SIAM Symposium on Discrete Algorithms. (2004)

[17] Engelfriet, J.: Top-down tree transducers with regular look-ahead. Math. Systems Theory **10** (1977) 289–303

[18] Engelfriet, J., Maneth, S.: A comparison of pebble tree transducers with macro tree transducers. Acta Informatica **39**(9) (2003) 613–698

[19] Maneth, S., Berlea, A., Perst, T., Seidl, H.: Xml type checking with macro tree transducers. In: PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, ACM (2005) 283–294

[20] Graehl, J., Knight, K., May, J.: Training tree transducers. Computational Linguistics **34**(3) (2008)

[21] Koller, A., Thater, S.: Computing weakest readings. In: Proceedings of the 48th ACL. (2010)

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 193-208
http://rewriting.loria.fr/rta/

ORDER-SORTED UNIFICATION WITH REGULAR EXPRESSION SORTS

TEMUR KUTSIA [1] AND MIRCEA MARIN [2]

[1] Research Institute for Symbolic Computation
Johannes Kepler University Linz, Austria
E-mail address: kutsia@risc.uni-linz.ac.at
URL: http://www.risc.uni-linz.ac.at/people/tkutsia/

[2] Graduate School of Systems and Information Engineering
University of Tsukuba, Japan
E-mail address: mmarin@cs.tsukuba.ac.jp
URL: http://www.score.is.tsukuba.ac.jp/~mmarin/

ABSTRACT. We extend first-order order-sorted unification by permitting regular expression sorts for variables and in the domains of function symbols. The set of basic sorts is finite. The obtained signature corresponds to a finite bottom-up hedge automaton. The unification problem in such a theory generalizes some known unification problems. Its unification type is infinitary. We give a complete unification procedure and prove decidability.

Introduction

In first-order order-sorted unification [Wal88], the set of basic sorts $\mathcal{B}$ is assumed to be partially ordered, variables are of basic sorts $\mathsf{s} \in \mathcal{B}$ and function symbols have sorts of the form $\mathsf{w} \to \mathsf{s}$, where w is a finite word over $\mathcal{B}$ and $\mathsf{s} \in \mathcal{B}$. We extend this framework by introducing regular expression sorts R over $\mathcal{B}$, allowing variables to be of sorts R and function symbols to have sorts $\mathsf{R} \to \mathsf{s}$. Another extension is that overloading function symbols is allowed. Under some reasonable conditions imposed over the signature [GM92], terms have the least sort.

Our signature has an interesting relation with automata. It is well-known that an order-sorted signature is a finite bottom-up tree automaton [Com89]. In our case, an order-sorted signature with regular expression sorts is exactly a finite bottom-up hedge automaton.

In this paper we study the unification problem for terms over an order-sorted signature with regular expression sorts. We call this problem *regular expression order-sorted unification* (REOSU) and show that it is infinitary, prove that it is decidable, and give a complete unification procedure.

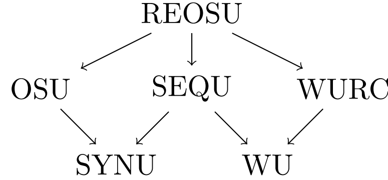
1998 ACM Subject Classification: F.4.1 [Theory of Computation]: Mathematical Logic and Formal Languages—Mathematical Logic, F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity—Nonnumerical Algorithms and Problems, F.4.3 [Theory of Computation]: Mathematical Logic and Formal Languages—Formal Languages.

Key words and phrases: Unification, sorts, regular expressions.

© T. Kutsia and M. Marin
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.193

REOSU extends some known problems as shown on the diagram below, illustrating its relations with syntactic unification (SYNU [Rob65]), word unification (WU [Sch90]), order-sorted unification (OSU [Wal88]), sequence unification (SEQU [Kut07]), and word unification with regular constraints (WURC [Sch90]):



Following the arrows, the problems are related as follows:

- From OSU one can obtain SYNU by restricting the sort hierarchy to be empty.
- SEQU problems without sequence variables (i.e., with individual variables only) constitute SYNU problems.
- WU is a special case of SEQU with constants, sequence variables, and only one flexible arity function symbol for concatenation.
- WU is also a special case of WURC where none of the variables are constrained.
- From REOSU we can get OSU (but with finitely many basic sort symbols only, because this is what REOSU considers) if instead of arbitrary regular sorts in function domains we allow only words over basic sorts, restrict variables to be of only basic sorts, and forbid function symbol overloading.
- SEQU can be obtained if we restrict REOSU with only one basic sort, say s, the variables that correspond to sequence variables in SEQU have the sort s^*, individual variables are of the sort s, and function symbols have the sort $\mathsf{s}^* \to \mathsf{s}$.
- WURC can be obtained from REOSU by the same restriction that gives WU from SEQU and, in addition, identifying the constants there to the corresponding sorts.

Order-sorted unification described in [SS89, Wei96] extends OSU from [Wal88] in a way that is not compatible with REOSU.

Regular expressions are presented in types in the programming language XDuce, designed for manipulating XML. These types are regular expressions over trees. They are ordered by a subtyping relation. Pattern matching for such regular expression types has been studied in [HP03]. Unlike XDuce types, our sorts are regular expressions over words and we perform word regular language manipulations rather than working with tree languages. Moreover, we are dealing with full-scale unification instead of matching.

In this paper we are dealing with REOSU in the empty theory (i.e., the syntactic case). It would also be interesting to see how one can extend equational OSU [Kir88, MGS89, Bou92, HM08] with regular expression sorts, but this problem is beyond the scope of this paper.

The paper is organized as follows. In Section 1 we give basic definitions and recall some known results. In Section 2 algorithms operating on sorts are given. Section 3 describes a complete unification procedure and discusses decidability. Section 4 concludes. Proofs can be found in the appendix.

For unification, we use the notation and terminology of [BS01]. For the notions related to sorted theories, we follow [GM92].

1. Preliminaries

Sorts

We consider a finite set $\mathcal{B}$ of basic sorts, partially ordered with the relation $\preceq$. Its elements are denoted with lowercase letters in $\mathsf{sans\ serif}$ font. $\mathsf{s} \prec \mathsf{r}$ means $\mathsf{s} \preceq \mathsf{r}$ and $\mathsf{s} \neq \mathsf{r}$. We write $\mathcal{R}$ for the set of regular expressions over $\mathcal{B}$, which is built in the standard way: $\mathsf{R} ::= \mathsf{s} \mid 1 \mid \mathsf{R}_1.\mathsf{R}_2 \mid \mathsf{R}_1{+}\mathsf{R}_2 \mid \mathsf{R}^*$. We use capital SANS SERIF font letters for them. The regular language denoted by a regular expression is: $[\![\mathsf{s}]\!] = \{\mathsf{s}\}$, $[\![1]\!] = \{\epsilon\}$, $[\![\mathsf{R}_1.\mathsf{R}_2]\!] = [\![\mathsf{R}_1]\!].[\![\mathsf{R}_2]\!]$, $[\![\mathsf{R}_1{+}\mathsf{R}_2]\!] = [\![\mathsf{R}_1]\!] \cup [\![\mathsf{R}_2]\!]$, $[\![\mathsf{R}^*]\!] = [\![\mathsf{R}]\!]^*$, where ϵ stands for the empty word, $[\![\mathsf{R}_1]\!].[\![\mathsf{R}_2]\!]$ is the concatenation of the regular languages $[\![\mathsf{R}_1]\!]$ and $[\![\mathsf{R}_2]\!]$, and $[\![\mathsf{R}]\!]^*$ is the Kleene star of $[\![\mathsf{R}]\!]$.

A *regular expression sort* is an element of $\mathcal{R}$, and a *functional expression sort* is an expression of the form $\mathsf{R} \to \mathsf{s}$ with $\mathsf{R} \in \mathcal{R}$ and $\mathsf{s} \in \mathcal{B}$. The relation $\preceq$ on $\mathcal{B}$ is extended to words of basic sorts, sets of words, and regular expression sorts as follows: (1) if $w_1, w_2 \in \mathcal{B}^*$ then $w_1 \preceq w_2$ iff $w_1 = \mathsf{s}_1 \cdots \mathsf{s}_n$, $w_2 = \mathsf{r}_1 \cdots \mathsf{r}_n$ and $\mathsf{s}_i \preceq \mathsf{r}_i$ for all $1 \leq i \leq n$; (2) if $W_1, W_2 \subseteq \mathcal{B}^*$ then $W_1 \preceq W_2$ iff for each $w_1 \in W_1$ there is $w_2 \in W_2$ such that $w_1 \preceq w_2$; and (3) if $\mathsf{R}_1, \mathsf{R}_2 \in \mathcal{R}$ then $\mathsf{R}_1 \preceq \mathsf{R}_2$ iff $[\![\mathsf{R}_1]\!] \preceq [\![\mathsf{R}_2]\!]$. Note that $\preceq$ is a quasi-order on the sets $\mathcal{B}$, $2^{\mathcal{B}^*}$, and $\mathcal{R}$. In particular, we can define the equivalence relation $\preceq$ on $\mathcal{R}$ by: $\mathsf{R}_1 \simeq \mathsf{R}_2$ iff $\mathsf{R}_1 \preceq \mathsf{R}_2$ and $\mathsf{R}_2 \preceq \mathsf{R}_1$. We extend this equivalence relation to functional sorts: $\mathsf{R}_1 \to \mathsf{s}_1 \simeq \mathsf{R}_2 \to \mathsf{s}_2$ iff $\mathsf{R}_1 \simeq \mathsf{R}_2$ and $\mathsf{s}_1 = \mathsf{s}_2$.

The *closure* $\overline{\mathsf{R}}$ of $\mathsf{R} \in \mathcal{R}$ is the regular expression defined as follows: $\overline{\mathsf{s}} = \sum_{\mathsf{r} \preceq \mathsf{s}} \mathsf{r}$, $\overline{1} = 1$, $\overline{\mathsf{R}_1.\mathsf{R}_2} = \overline{\mathsf{R}}_1.\overline{\mathsf{R}}_2$, $\overline{\mathsf{R}_1{+}\mathsf{R}_2} = \overline{\mathsf{R}}_1{+}\overline{\mathsf{R}}_2$, $\overline{\mathsf{R}^*} = \overline{\mathsf{R}}^*$. Closures of regular expressions enable the decidability of relations $\preceq$ and $\simeq$ on $\mathcal{R}$:

Lemma 1. *Let* $\mathsf{S}, \mathsf{R} \in \mathcal{R}$. *Then* $\mathsf{S} \preceq \mathsf{R}$ *iff* $[\![\overline{\mathsf{S}}]\!] \subseteq [\![\overline{\mathsf{R}}]\!]$.

Corollary 1. *Let* $\mathsf{S}, \mathsf{R} \in \mathcal{R}$. *Then* $\mathsf{S} \simeq \mathsf{R}$ *iff* $[\![\overline{\mathsf{S}}]\!] = [\![\overline{\mathsf{R}}]\!]$.

The set of all $\preceq$-maximal elements of a set of sorts $S \subseteq \mathcal{R}$ is denoted $\max(S)$. R is a lower bound of S if $\mathsf{R} \preceq \mathsf{Q}$ for all $\mathsf{Q} \in S$. A lower bound G of S is a greatest lower bound, denoted $\mathrm{glb}(S)$, if $\mathsf{R} \preceq \mathsf{G}$ for all lower bounds R of S. Note that if $\mathrm{glb}(S)$ exists, then it is unique modulo $\simeq$.

Terms

For each R we assume a countable set of variables $\mathcal{V}_{\mathsf{R}}$ such that $\mathcal{V}_{\mathsf{R}_1} = \mathcal{V}_{\mathsf{R}_2}$ iff $\mathsf{R}_1 \simeq \mathsf{R}_2$ and $\mathcal{V}_{\mathsf{R}_1} \cap \mathcal{V}_{\mathsf{R}_2} = \emptyset$ if $\mathsf{R}_1 \not\simeq \mathsf{R}_2$. Also, for each $\mathsf{R} \in \mathcal{R}, \mathsf{s} \in \mathcal{B}$ we assume a set of function symbols $\mathcal{F}_{\mathsf{R} \to \mathsf{s}}$ such that $\mathcal{F}_{\mathsf{R}_1 \to \mathsf{s}_1} = \mathcal{F}_{\mathsf{R}_2 \to \mathsf{s}_2}$ iff $\mathsf{R}_1 \to \mathsf{s}_1 \simeq \mathsf{R}_2 \to \mathsf{s}_2$. Moreover, the following conditions should be satisfied:

Preregularity: If $f \in \mathcal{F}_{\mathsf{R}_1 \to \mathsf{s}_1}$ and $\mathsf{R}_2 \preceq \mathsf{R}_1$, then there is a $\preceq$-least element in the set $\{\mathsf{s} \mid f \in \mathcal{F}_{\mathsf{R} \to \mathsf{s}} \text{ and } \mathsf{R}_2 \preceq \mathsf{R}\}$.

Finite overloading: For each f, the set $\{\mathcal{F}_{\mathsf{R} \to \mathsf{s}} \mid \mathsf{R} \in \mathcal{R}, \mathsf{s} \in \mathcal{B}, f \in \mathcal{F}_{\mathsf{R} \to \mathsf{s}}\}$ is finite.

We say that R is a sort of x if $x \in \mathcal{V}_{\mathsf{R}}$. Similarly, $\mathsf{R} \to \mathsf{s}$ is a sort of f if $f \in \mathcal{F}_{\mathsf{R} \to \mathsf{s}}$. Function symbols from $\mathcal{F}_{1 \to \mathsf{s}}$ are called constants. We use the letters a, b, c to denote them. We will write $f : \mathsf{R} \to \mathsf{s}$ for $f \in \mathcal{F}_{\mathsf{R} \to \mathsf{s}}$, $a : \mathsf{s}$ for $a \in \mathcal{F}_{1 \to \mathsf{s}}$, and $x : \mathsf{R}$ for $x \in \mathcal{V}_{\mathsf{R}}$. Setting $\mathcal{V} = \cup_{\mathsf{R} \in \mathcal{R}} \mathcal{V}_{\mathsf{R}}$ and $\mathcal{F} = \cup_{\mathsf{R} \in \mathcal{R}, \mathsf{s} \in \mathcal{B}} \mathcal{F}_{\mathsf{R} \to \mathsf{s}}$, we define the sets $\mathcal{T}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$ of *terms of sort* $\mathsf{R} \in \mathcal{R}$ *over* $\mathcal{V}$ *and* $\mathcal{F}$, and $\mathcal{TS}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$ of *term sequences of sort* $\mathsf{R} \in \mathcal{R}$ *over* $\mathcal{V}$ *and* $\mathcal{F}$, as the least sets satisfying the properties:

- $\mathcal{V}_{\mathsf{R}} \subseteq \mathcal{T}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$.
- $\mathcal{T}_{\mathsf{R}'}(\mathcal{F}, \mathcal{V}) \subseteq \mathcal{T}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$ if $\mathsf{R}' \preceq \mathsf{R}$.
- $\epsilon \in \mathcal{TS}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$ if $1 \preceq \mathsf{R}$.
- $(t_1, \ldots, t_n) \in \mathcal{TS}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$ if there exist $\mathsf{R}_1, \ldots, \mathsf{R}_n \in \mathcal{R}$ such that $t_i \in \mathcal{T}_{\mathsf{R}_i}(\mathcal{F}, \mathcal{V})$ and $\mathsf{R}_1.\cdots.\mathsf{R}_n \preceq \mathsf{R}$.
- $f(\tilde{t}) \in \mathcal{T}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$, if $\mathsf{R} = \mathsf{s}$, $f : \mathsf{R}' \to \mathsf{s}$, and $\tilde{t} \in \mathcal{TS}_{\mathsf{R}'}(\mathcal{F}, \mathcal{V})$.

The set of terms over $\mathcal{V}$ and $\mathcal{F}$ is defined as $\mathcal{T}(\mathcal{F}, \mathcal{V}) = \cup_{\mathsf{R} \in \mathcal{R}} \mathcal{T}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$. We abbreviate terms $a(\epsilon)$ with a. The *depth* of a term and a term sequence is defined in the standard way: $\mathit{depth}(x) = 1$, $\mathit{depth}(f(\tilde{t})) = 1 + \mathit{depth}(\tilde{t})$, $\mathit{depth}(\epsilon) = 0$, $\mathit{depth}(t_1, \ldots, t_n) = \max\{\mathit{depth}(t_i) \mid 1 \le i \le n\}$, $n > 0$.

Lemma 2. *Every term has a $\preceq$-least sort R that is unique modulo $\simeq$.*

The $\preceq$-least sort of a term t modulo $\simeq$ is called the *least sort* of t, and is denoted by $\mathit{lsort}(t)$. In the same way, the $\preceq$-least sort of a term sequence $(t_1, \ldots, t_n)$, $n \ge 1$, is defined uniquely modulo $\simeq$ as $\mathit{lsort}(t_1).\cdots.\mathit{lsort}(t_n)$ and is denoted by $\mathit{lsort}(t_1, \ldots, t_n)$. When $n = 0$, i.e., for the empty sequence, $\mathit{lsort}(\epsilon) = 1$.

The set of variables of a term t is denoted by $\mathit{var}(t)$. A term t is ground if $\mathit{var}(t) = \emptyset$. These notions extend to term sequences, sets of term sequences, etc.

For a basic sort s, its semantics $\mathit{sem}(\mathsf{s})$ is the set $\mathcal{T}_{\mathsf{s}}(\mathcal{F})$ of ground terms of sort s. The semantics of a regular sort is given by the set of ground term sequences of the corresponding sort: $\mathit{sem}(1) = \{\epsilon\}$, $\mathit{sem}(\mathsf{R}_1.\mathsf{R}_2) = \{(\tilde{s}_1, \tilde{s}_2) \mid \tilde{s}_1 \in \mathit{sem}(\mathsf{R}_1), \tilde{s}_2 \in \mathit{sem}(\mathsf{R}_2)\}$, $\mathit{sem}(\mathsf{R}_1{+}\mathsf{R}_2) = \mathit{sem}(\mathsf{R}_1) \cup \mathit{sem}(\mathsf{R}_2)$, $\mathit{sem}(\mathsf{R}^*) = \mathit{sem}(\mathsf{R})^*$. This definition, together with the definition of $\preceq$ and $\mathcal{T}_{\mathsf{R}}(\mathcal{F}, \mathcal{V})$, implies that if $\mathsf{R} \preceq \mathsf{Q}$, then $\mathit{sem}(\mathsf{R}) \subseteq \mathit{sem}(\mathsf{Q})$.

Substitutions and Unification Problems

A *substitution* is a well-sorted mapping from variables to term sequences, which is identity almost everywhere. Substitutions are denoted with lowercase Greek letters, where ε stands for the identity substitution. Well-sortedness of σ means that $\mathit{lsort}(\sigma(x)) \preceq \mathit{lsort}(x)$ for all x. The notions of substitution application, term and term sequence instances, substitution composition, restriction, and subsumption are defined in the standard way. We use postfix notation for instances, juxtaposition for composition, and write $\sigma \le_{\mathcal{X}} \vartheta$ for subsumption meaning that σ is more general than ϑ on the set of variables $\mathcal{X}$. The *depth* of a substitution is defined as $\mathit{depth}(\sigma) = \max\{\mathit{depth}(x\sigma) \mid x \in \mathcal{V}\}$.

Lemma 3. *$\mathit{lsort}(t\sigma) \preceq \mathit{lsort}(t)$ and $\mathit{lsort}(\tilde{t}\sigma) \preceq \mathit{lsort}(\tilde{t})$ hold for any term t, term sequence $\tilde{t}$ and substitution σ.*

An *equation* is a pair of term sequences, written as $\tilde{s} \doteq \tilde{t}$. Its *depth* is the maximum between $\mathit{depth}(\tilde{s})$ and $\mathit{depth}(\tilde{t})$. A *regular expression order sorted unification* or, shortly, REOSU problem Γ is a finite set of equations between sorted term sequences $\{\tilde{s}_1 \doteq \tilde{t}_1, \ldots, \tilde{s}_n \doteq \tilde{t}_n\}$. A substitution σ is a *unifier* of Γ if $\tilde{s}_i\sigma = \tilde{t}_i\sigma$ for all $1 \le i \le n$. A *minimal complete set* of unifiers of Γ is a set U of unifiers of Γ satisfying the following conditions:

Completeness: For any unifier ϑ of Γ there is $\sigma \in U$ such that $\sigma \le_{\mathit{var}(\Gamma)} \vartheta$.
Minimality: If there are $\sigma_1, \sigma_2 \in U$ such that $\sigma_1 \le_{\mathit{var}(\Gamma)} \sigma_2$, then $\sigma_1 = \sigma_2$.

The *depth* of a REOSU problem Γ is the maximum depth of the equations it contains.

Linear Form and Split of a Regular Expression

We recall the notion of linear form for regular expressions from [Ant96] by adapting the notation to our setting and using the set of basic sorts $\mathcal{B}$ for alphabet. This notion, together with the split of a regular expression, will be needed later, in sort-related algorithms: When we decompose a hedge in the weakening process, we have to split the corresponding sort as well. Linear form helps to split a sort into a basic sort and another sort, while the split operation decomposes it into two (not necessarily basic) sorts.

A pair (s, R) is called a *monomial*. A *linear form* of a regular expression R, denoted $\mathit{lf}(\mathsf{R})$, is a finite set of monomials defined recursively as follows:

$$
\begin{array}{rclrcl}
\mathit{lf}(1) & = & \emptyset & \mathit{lf}(\mathsf{R}^*) & = & \mathit{lf}(\mathsf{R}) \odot \mathsf{R}^* \\
\mathit{lf}(\mathsf{s}) & = & \{(\mathsf{s}, 1)\} & \mathit{lf}(\mathsf{R.Q}) & = & \mathit{lf}(\mathsf{R}) \odot \mathsf{Q} \quad \text{if } \epsilon \notin [\![\mathsf{R}]\!] \\
\mathit{lf}(\mathsf{s{+}r}) & = & \mathit{lf}(\mathsf{s}) \cup \mathit{lf}(\mathsf{r}) & \mathit{lf}(\mathsf{R.Q}) & = & \mathit{lf}(\mathsf{R}) \odot \mathsf{Q} \cup \mathit{lf}(\mathsf{Q}) \quad \text{if } \epsilon \in [\![\mathsf{R}]\!]
\end{array}
$$

These equations involve an extension of concatenation $\odot$ that acts on a linear form and a regular expression and returns a linear form. It is defined as $l \odot 1 = l$ and $l \odot \mathsf{Q} = \{(\mathsf{s}, \mathsf{S.Q}) \mid (\mathsf{s}, \mathsf{S}) \in l, \mathsf{S} \neq 1\} \cup \{(\mathsf{s}, \mathsf{Q}) \mid (\mathsf{s}, 1) \in l\}$ if $\mathsf{Q} \neq 1$.

As an example, $\mathit{lf}(\mathsf{R}) = \{(\mathsf{s}, \mathsf{R}), (\mathsf{s}, \mathsf{s}.(\mathsf{s.s{+}r})^*), (\mathsf{r}, (\mathsf{s.s{+}r})^*)\}$ for $\mathsf{R} = \mathsf{s}^*.(\mathsf{s.s{+}r})^*$. The set $\hat{\mathit{lf}}(\mathsf{R})$ is defined as $\{\mathsf{s.Q} \mid (\mathsf{s}, \mathsf{Q}) \in \mathit{lf}(\mathsf{R})\}$.

Definition 1 (Split)**.** Let $\mathsf{S} \in \mathcal{R}$. A *split* of S is a pair $(\mathsf{Q}, \mathsf{R}) \in \mathcal{R}^2$ such that (1) $\mathsf{Q.R} \preceq \mathsf{S}$ and (2) if $(\mathsf{Q}', \mathsf{R}') \in \mathcal{R}^2$, $\mathsf{Q} \preceq \mathsf{Q}'$, $\mathsf{R} \preceq \mathsf{R}'$, and $\mathsf{Q}'.\mathsf{R}' \preceq \mathsf{S}$, then $\mathsf{Q} \simeq \mathsf{Q}'$ and $\mathsf{R} \simeq \mathsf{R}'$.

We recall the definition of 2-factorization from [Con71]: A pair $(\mathsf{Q}, \mathsf{R}) \in \mathcal{R}^2$ is a *2-factorization* of $\mathsf{S} \in \mathcal{R}$ if (1) $[\![\mathsf{Q.R}]\!] \subseteq [\![\mathsf{S}]\!]$ and (2) if $(\mathsf{Q}', \mathsf{R}') \in \mathcal{R}^2$, $[\![\mathsf{Q}]\!] \subseteq [\![\mathsf{Q}']\!]$, $[\![\mathsf{R}]\!] \subseteq [\![\mathsf{R}']\!]$, and $[\![\mathsf{Q}'.\mathsf{R}']\!] \subseteq [\![\mathsf{S}]\!]$, then $[\![\mathsf{Q}]\!] = [\![\mathsf{Q}']\!]$ and $[\![\mathsf{R}]\!] = [\![\mathsf{R}']\!]$.

Lemma 4. (Q, R) *is a split of* S *iff* $(\overline{\mathsf{Q}}, \overline{\mathsf{R}})$ *is a 2-factorization of* $\overline{\mathsf{S}}$.

In [Con71] it has been shown that the 2-factorizations of a regular expression are finitely many modulo $\simeq$, and that they can be effectively computed. By the lemma above a regular expression has finitely many splits modulo $\simeq$ that can be effectively computed. For instance, the regular expression $\mathsf{s}^*.\mathsf{r}.\mathsf{r}^*$ has two splits modulo $\simeq$: $(\mathsf{s}^*, \mathsf{s}^*.\mathsf{r}.\mathsf{r}^*)$ and $(\mathsf{s}^*.\mathsf{r}.\mathsf{r}^*, \mathsf{r}^*)$.

Relating REOS Signatures and Hedge Automata

Regular expression ordered sorts are related to regular hedge automata in the same way as ordered sorts are related to tree automata. Namely, a REOS signature is a finite bottom-up hedge automaton.

To illustrate this relation, we first recall the definition of nondeterministic finite hedge automaton (NFHA) from [CDG+]: An NFHA over Σ is a tuple (Q, Σ, Q_f, Δ) where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a finite set of transition rules of the following types:

- $a(R) \to q$ where $a \in \Sigma$, $q \in Q$, and $R \subseteq Q^*$ is a regular language over Q, or
- $q' \to q$ (called ϵ-transitions), where $q', q \in Q$.

Now, we can take our set of basic sorts $\mathcal{B}$ in the role of Q, the set $\mathcal{F}$ in the role of Σ, assume $Q_f = Q$, and define Δ as follows: For each $\mathsf{r} \prec \mathsf{s}$, the ϵ-transition rule $\mathsf{r} \to \mathsf{s}$ is in Δ. For each $f : \mathsf{R} \to \mathsf{s}$, the rule $f(\mathsf{R}) \to \mathsf{s}$ is also in Δ. It is easy to see that our ground terms are exactly the unranked trees recognized by this automaton. A ground term of sort s is an unranked tree recognized by the automaton at state s.

2. Sort-Related Algorithms

In this section we identify algorithms to decide $\preceq$ on $\mathcal{R}$, to compute the greatest lower bounds for regular expression sorts, and to compute sort-weakening substitutions.

Deciding $\preceq$

Without an ordering on basic sorts, $\preceq$ would be the standard inequality for regular word expressions which can be decided, for instance, by Antimirov's algorithm [Ant95] that employs partial derivatives. The problem is PSPACE-complete, but this rewriting approach has an advantage over the standard technique of translating regular expressions into automata: With it, in some cases solving derivations can have polynomial size, while any algorithm based on translation of regular expressions into DFA's causes an exponential blow-up.

In our case, we can rely on the property that $\mathsf{S} \preceq \mathsf{R}$ iff $[\![\overline{\mathsf{S}}]\!] \subseteq [\![\overline{\mathsf{R}}]\!]$, proved in Lemma 1. The property $[\![\overline{\mathsf{S}}]\!] \subseteq [\![\overline{\mathsf{R}}]\!]$ can be decided by Antimirov's original algorithm on $\overline{\mathsf{S}}$ and $\overline{\mathsf{R}}$.

Computing Greatest Lower Bounds

A greatest lower bound of regular expressions would be their intersection, if we did not have ordering on the basic sorts. Intersection can be computed either in the standard way, by translating them into automata, or by Antimirov & Mosses's rewriting algorithm [AM95] for regular expressions extended with the intersection operator. Computation requires double exponential time.

Here we can employ the regular expression intersection algorithm [AM95] to compute a greatest lower bound, with one modification: To compute the intersection between two alphabet letters (i.e. between two basic sorts), instead of standard check whether they are the same, we compute the maximal elements in the set of their lower bounds. There can be several such maximal elements. This can be easily computed based on the ordering on basic sorts. Then we can take the sum of these elements and it will be a greatest lower bound. This construction allows to compute a greatest lower bound of two regular expressions, which is unique modulo $\simeq$.

An implementation of Antimirov-Mosses algorithm [Sul09] requires only minor modifications to deal with the ordering on alphabet letters (basic sorts). Hence, for S and R we compute here $\mathrm{glb}(\mathsf{S},\mathsf{R})$ and we know that if Q is a regular expression with $[\![\mathsf{Q}]\!] = [\![\overline{\mathsf{S}}]\!] \cap [\![\overline{\mathsf{R}}]\!]$, then $\mathrm{glb}(\mathsf{S},\mathsf{R}) \simeq \mathsf{Q}$.[1]

Computing Weakening Substitutions

Now we describe an algorithm that computes a substitution to weaken the sort of a term sequence towards a given sort. The necessity of such an algorithm can be demonstrated on a simple example: Assume we want to unify x and $f(y)$ for $x : \mathsf{s}$, $f : \mathsf{R}_1 \to \mathsf{s}_1$, $f : \mathsf{R}_2 \to \mathsf{s}_2$, $y : \mathsf{R}_2$, where $\mathsf{s}_1 \prec \mathsf{s} \prec \mathsf{s}_2$ and $\mathsf{R}_1 \prec \mathsf{R}_2$. We can not unify x with $f(y)$ directly, because $\mathit{lsort}(f(y)) = \mathsf{s}_2 \not\preceq \mathsf{s} = \mathit{lsort}(x)$. However, if we weaken the least sort of $f(y)$ to s_1, then unification is possible. To weaken the least sort of $f(y)$, we take its instance under

[1]We say that the computation of glb fails, if the (modification of) Antimirov-Mosses algorithm returns 0, and express it as $\mathrm{glb}(\mathsf{S},\mathsf{R}) = \bot$.

substitution $\{y \mapsto z\}$, where $z \in \mathcal{V}_{\mathsf{R}_1}$, which gives $lsort(f(z)) = \mathsf{s}_1$. Hence, the substitution $\{y \mapsto z, x \mapsto f(z)\}$ is a unifier of x and $f(y)$, leading to the common instance $f(z)$.

A *weakening pair* is a pair of a term sequence $\tilde{t}$ and a sort Q, written $\tilde{t} \rightsquigarrow \mathsf{Q}$. A substitution ω is called a *weakening substitution* of a set W of weakening pairs iff $lsort(\tilde{t}\omega) \preceq \mathsf{Q}$ for each $\tilde{t} \rightsquigarrow \mathsf{Q} \in W$.

Our weakening algorithm is called $\mathfrak{W}$, and works by applying exhaustively the following rules to pairs of the form $W; \sigma$ where W is a set of weakening pairs and σ is a substitution:

R-w: **Remove a Weakening Pair**
$\{\tilde{t} \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow W; \sigma \qquad$ if $lsort(\tilde{t}) \preceq \mathsf{Q}$.

D1-w: **Decomposition 1 in Weakening**
$\{(f(\tilde{t}), \tilde{s}) \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow \{f(\tilde{t}) \rightsquigarrow \mathsf{s}, \tilde{s} \rightsquigarrow \mathsf{S}\} \cup W; \sigma$
if $lsort(f(\tilde{t}), \tilde{s}) \not\preceq \mathsf{Q}$, $var(f(\tilde{t}), \tilde{s}) \neq \emptyset$, $\tilde{s} \neq \epsilon$ and $\mathsf{s}.\mathsf{S} \in \max(\hat{lf}(\mathsf{Q}))$.

D2-w: **Decomposition 2 in Weakening**
$\{(x, \tilde{s}) \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow \{x \rightsquigarrow \mathsf{Q}_1, \tilde{s} \rightsquigarrow \mathsf{Q}_2\} \cup W; \sigma$
if $lsort(x, \tilde{s}) \not\preceq \mathsf{Q}$, $\tilde{s} \neq \epsilon$ and $(\mathsf{Q}_1, \mathsf{Q}_2)$ is a split of Q.

AS-w: **Argument Sequence Weakening**
$\{f(\tilde{t}) \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow \{\tilde{t} \rightsquigarrow \mathsf{R}\} \cup W; \sigma$
where $lsort(f(\tilde{t})) \not\preceq \mathsf{Q}$, $var(f(\tilde{t})) \neq \emptyset$, $\mathsf{R}.\mathsf{r}$ is a maximal sort such that $f \in \mathcal{F}_{\mathsf{R} \to \mathsf{r}}$ and $\mathsf{r} \preceq \mathsf{Q}$.

V-w: **Variable Weakening**
$\{x \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow W\sigma; \sigma\{x \mapsto w\}$
where $\mathrm{glb}(\{lsort(x), \mathsf{Q})\}) \neq \bot$ and w is a fresh variable from $\mathcal{V}_{\mathrm{glb}(\{lsort(x), \mathsf{Q})\})}$.

If none of the rules are applicable to $W; \sigma$, then it is transformed into $\bot$, indicating failure. By exhaustive search, transforming each $W; \sigma$ in all possible ways, we generate a complete search tree whose branches form *derivations*. The branches that end with $\bot$ are called failing branches. The branches that end with $\emptyset; \omega$ are called successful branches and ω is a substitution computed by $\mathfrak{W}$ along this branch. The set of all substitutions computed by $\mathfrak{W}$ starting from $W; \epsilon$ is denoted by $weak(W)$. It is easy to see that the elements of $weak(W)$ are variable renaming substitutions.

It is essential that the signature has the finite overloading property, which guarantees that the rule AS-w does not introduce infinite branching. Since the linear form and split of a regular expression are both finite, the other rules do not cause infinite branching either. $\mathfrak{W}$ is terminating, sound, and complete, as the following theorems show.

Theorem 1. *$\mathfrak{W}$ is terminating.*

Theorem 2 (Soundness of the Weakening Algorithm)**.** *Each $\omega \in weak(W)$ is a weakening substitution of W.*

Theorem 3 (Completeness of the Weakening Algorithm)**.** *For every weakening substitution ω of W there exists $\omega' \in weak(W)$ such that $\omega' \leq_{var(W)} \omega$.*

Example 1. Let $W = \{x \rightsquigarrow \mathsf{q}, f(x) \rightsquigarrow \mathsf{s}\}$ be a weakening problem with $x : \mathsf{r}$, $f : \mathsf{s} \to \mathsf{s}$, $f : \mathsf{r} \to \mathsf{r}$ and the sorts $\mathsf{r}_1 \prec \mathsf{r}$, $\mathsf{r}_2 \prec \mathsf{r}$, $\mathsf{r}_1 \prec \mathsf{q}$, $\mathsf{r}_2 \prec \mathsf{q}$, $\mathsf{s} \prec \mathsf{r}_1$, $\mathsf{s} \prec \mathsf{r}_2$. Then the weakening algorithm first transforms $W; \varepsilon$ into $\{f(w) \rightsquigarrow \mathsf{s}\}; \{x \mapsto w\}$ with $w : \mathsf{r}_1{+}\mathsf{r}_2$ by the rule V-w.

The obtained weakening pair is then transformed into $\emptyset; \{\{x \mapsto z, w \mapsto z\}\}$ with $z : \mathsf{s}$ by $\mathsf{AS\text{-}w}$, leading to $\mathit{weak}(W) = \{\{x \mapsto z\}\}$.

Example 2. Let $W = \{(x, y) \rightsquigarrow \mathsf{s}^*.\mathsf{r}.\mathsf{r}^*\}$ be a weakening problem with $x : \mathsf{q}_1^*.\mathsf{p}_1^*$, $y : \mathsf{q}_2^*.\mathsf{p}_2^*$, and the sorts $\mathsf{s} \prec \mathsf{q}_1$, $\mathsf{s} \prec \mathsf{q}_2$, $\mathsf{r} \prec \mathsf{p}_1$, $\mathsf{r} \prec \mathsf{p}_2$. Then the weakening algorithm computes $\mathit{weak}(W) = \{\{x \mapsto u_1, y \mapsto v_1\}, \{x \mapsto u_2, y \mapsto v_2\}\}$ where $u_1 : \mathsf{s}^*.\mathsf{r}.\mathsf{r}^*$, $v_1 : \mathsf{r}^*$, $u_2 : \mathsf{s}^*$ and $v_2 : \mathsf{s}^*.\mathsf{r}.\mathsf{r}^*$.

Example 3. Let $W = \{x \rightsquigarrow \mathsf{q}^*\}$ be a weakening problem with $x : \mathsf{r}^*$ and the sorts $\mathsf{s}_1 \prec \mathsf{r}$, $\mathsf{s}_2 \prec \mathsf{r}$, $\mathsf{s}_1 \prec \mathsf{q}$, $\mathsf{s}_2 \prec \mathsf{q}$, $\mathsf{p}_1 \prec \mathsf{s}_1$, $\mathsf{p}_2 \prec \mathsf{s}_2$. Then the weakening algorithm computes $\mathit{weak}(W) = \{\{x \mapsto w\}\}$ where $w : (\mathsf{s}_1{+}\mathsf{s}_2)^*$.

3. Unification Type, Unification Procedure, Decidability

Unification Type

Let Γ_{re} be a REOSU problem and Γ_{seq} its version without sorts, i.e. a SEQU problem. Each unifier of Γ_{re} is either a unifier of Γ_{seq} or is obtained from a unifier of Γ_{seq} by composing it with a weakening substitution as follows: If $\sigma = \{x_1 \mapsto \tilde{t}_1, \dots, x_n \mapsto \tilde{t}_n\}$ is a unifier of Γ_{seq}, then the set of weakening substitutions for σ is $\Omega(\sigma) = \mathit{weak}(\{\tilde{t}_1 \rightsquigarrow \mathit{lsort}(x_1), \dots, \tilde{t}_n \rightsquigarrow \mathit{lsort}(x_n)\})$. For each $\omega_\sigma \in \Omega(\sigma)$, $\sigma\omega_\sigma$ is a unifier of Γ_{re}. Since SEQU is infinitary, the type of REOSU can be either infinitary or nullary, and we show now that it is not nullary.

Let S_{seq} be a minimal complete set of unifiers of Γ_{seq} and S_{re} be the set containing the unifiers of Γ_{re} that are either in S_{seq} or are obtained by weakening unifiers in S_{re}. Since $\{\sigma\omega_\sigma \mid \omega_\sigma \in \Omega(\sigma)\}$ is finite for each σ, we can assume that S_{re} contains only a minimal subset of it for each σ. The set S_{re} is complete. Assume by contradiction that it is not minimal. Then it contains σ' and ϑ' such that $\sigma' \leq_{\mathit{var}(\Gamma_{\mathrm{re}})} \vartheta'$, i.e., there exists φ' such that $\sigma'\varphi' =_{\mathit{var}(\Gamma_{\mathrm{re}})} \vartheta'$. If $\vartheta' \in S_{\mathrm{seq}}$, then we have $\sigma'\varphi' = \sigma\omega_\sigma\varphi' =_{\mathit{var}(\Gamma)} \vartheta'$ for an $\omega_\sigma \in \Omega(\sigma)$, which contradicts minimality of S_{seq}. If $\sigma' \in S_{\mathrm{seq}}$, then $\sigma'\varphi' =_{\mathit{var}(\Gamma_{\mathrm{re}})} \vartheta' = \vartheta\omega_\vartheta$ where $\omega_\vartheta \in \Omega(\vartheta)$. Since ω_ϑ is variable renaming, $\sigma'\varphi'\omega_\vartheta^{-1} =_{\mathit{var}(\Gamma_{\mathrm{seq}})} \vartheta$, which again contradicts minimality of S_{seq}. Both σ' and ϑ' can not be from S_{seq} because S_{seq} is minimal. If neither σ' nor ϑ' is in S_{seq}, then we have $\sigma\omega_\sigma\varphi' = \sigma'\varphi' =_{\mathit{var}(\Gamma_{\mathrm{re}})} \vartheta' = \vartheta\omega_\vartheta$ and again a contradiction: $\sigma\omega_\sigma\varphi'\omega_\vartheta^{-1} =_{\mathit{var}(\Gamma_{\mathrm{seq}})} \vartheta$.

Hence, for any Γ_{re} there is a complete set of unifiers with no two elements comparable with respect to $\leq_{\mathit{var}(\Gamma_{\mathrm{re}})}$, which implies that Γ_{re} has a minimal complete set of unifiers and REOSU is not nullary.

Unification Procedure

To compute unifiers for a REOSU problem, one way is, first, to ignore the sort information, employ the SEQU procedure [Kut02, Kut07] on the unsorted problem, and then weaken each computed substitution to obtain their order-sorted instances. In fact, such an approach is not uncommon in order-sorted unification, see, e.g. [SS89, MGS89, SNGM89, HM08]. It has an advantage of being a modular method that reuses an existing solving procedure.

In our case, this approach can be realized as follows: Assume a SEQU procedure computes a unifier $\sigma = \{x_1 \mapsto \tilde{t}_1, \dots, x_n \mapsto \tilde{t}_n\}$[2] of the unsorted version of an REOSU problem Γ.

[2]We assume without loss of generality that σ is idempotent.

Then we form a set of weakening pairs $W = \{\tilde{t}_1 \rightsquigarrow \mathsf{Q}_1, \dots, \tilde{t}_n \rightsquigarrow \mathsf{Q}_n\}$, where the Q's are the sorts of the corresponding x's, and find the set of weakening substitutions $\mathit{weak}(W)$. If $\mathit{weak}(W) = \emptyset$, then σ can not be weakened further to a solution of Γ. Otherwise, $\sigma\vartheta$ is a solution of Γ for each $\vartheta \in \mathit{weak}(W)$.

A drawback of this approach is that it is so called generate-and-test method. It is not able to detect early enough derivations that fail because of sort incompatibility. Early failure detection requires weakening to be tailored in the unification rules. This is what we consider in more details now.

The following transformation rules act on pairs of the form $\Gamma; \sigma$ with Γ a unification problem and σ a substitution, and are designed to define a sound and complete rule-based procedure for REOSU problems.

P: **Projection**

$\Gamma; \sigma \Longrightarrow \Gamma\vartheta; \sigma\vartheta,$

for $\vartheta = \{x_1 \mapsto \epsilon, \dots, x_n \mapsto \epsilon\}$ with $x_i \in \mathit{var}(\Gamma)$ and $1 \preceq \mathit{lsort}(x_i)$ for $1 \le i \le n$.

T: **Trivial**

$\{\tilde{t} \doteq \tilde{t}\} \cup \Gamma; \sigma \Longrightarrow \Gamma; \sigma.$

TP: **Trivial Prefix**

$\{(\tilde{r}, \tilde{t}) \doteq (\tilde{r}, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}\} \cup \Gamma; \sigma, \qquad$ if $\tilde{r} \neq \epsilon$ and $\tilde{t} \neq \tilde{s}$.

D: **Decomposition**

$\{(f(\tilde{t}), \tilde{t}') \doteq (f(\tilde{s}), \tilde{s}')\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}, \tilde{t}' \doteq \tilde{s}'\} \cup \Gamma; \sigma,$

if $\mathrm{glb}(\{\mathit{lsort}(f(\tilde{t})), \mathit{lsort}(f(\tilde{s}))\}) \neq \bot$ and $\tilde{t} \neq \tilde{s}$.

O: **Orient**

$\{(t, \tilde{t}) \doteq (x, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(x, \tilde{s}) \doteq (t, \tilde{t})\} \cup \Gamma; \sigma, \qquad$ where $t \notin \mathcal{V}$.

WkE1: **Weakening and Elimination 1**

$\{(x, \tilde{t}) \doteq (s, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta,$

where $s \notin \mathcal{V}$, $x \notin \mathit{var}(s)$, $\omega \in \mathit{weak}(\{s \rightsquigarrow \mathit{lsort}(x)\})$, and $\vartheta = \omega \cup \{x \mapsto s\omega\}$.

WkE2: **Weakening and Elimination 2**

$\{(x, \tilde{t}) \doteq (y, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq \tilde{s}\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta,$

where $\mathsf{R} = \mathrm{glb}(\mathit{lsort}(x), \mathit{lsort}(y)) \not\preceq 1$ and $\vartheta = \{x \mapsto w, y \mapsto w\}$ for a fresh variable $w \in \mathcal{V}_{\mathsf{R}}$.

WkWd1: **Weakening and Widening 1**

$\{(x, \tilde{t}) \doteq (s, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(z, \tilde{t}) \doteq \tilde{s}\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta,$

if $s \notin \mathcal{V}$, $x \notin \mathit{var}(s)$, there is $(\mathsf{r}, \mathsf{R}) \in \mathit{lf}(\mathit{lsort}(x))$ with $\mathsf{R} \not\preceq 1$, $\omega \in \mathit{weak}(\{s \rightsquigarrow \mathsf{r}\})$, $z \in \mathcal{V}_{\mathsf{R}}$ is a fresh variable and $\vartheta = \omega \cup \{x \mapsto (s\omega, z)\}$.

WkWd2: **Weakening and Widening 2**

$\{(x, \tilde{t}) \doteq (y, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{(z, \tilde{t}) \doteq \tilde{s}\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta,$

where (S, R) is a split of $\mathit{lsort}(x)$ such that $\mathsf{R} \not\preceq 1$, $w \in \mathcal{V}_{\mathsf{R}'}$ is a fresh variable with $\mathsf{R}' = \mathrm{glb}(\{\mathsf{S}, \mathit{lsort}(y)\}) \not\preceq 1$, z is a fresh variable with $\mathit{lsort}(z) = \mathsf{R}$, and $\vartheta = \{x \mapsto (w, z), y \mapsto w\}$.

WkWd3: Weakening and Widening 3

$\{(x, \tilde{t}) \doteq (y, \tilde{s})\} \cup \Gamma; \sigma \Longrightarrow \{\tilde{t} \doteq (z, \tilde{s})\}\vartheta \cup \Gamma\vartheta; \sigma\vartheta$,

where (S, R) is a split of $lsort(y)$ such that $\mathsf{R} \not\simeq 1$, $w \in \mathcal{V}_{\mathsf{R}'}$ is a fresh variable with $\mathsf{R}' = \mathrm{glb}(\{\mathsf{S}, lsort(x)\}) \not\simeq 1$, z is a fresh variable with $lsort(z) = \mathsf{R}$, and $\vartheta = \{x \mapsto w, y \mapsto (w, z)\}$.

Note that $\mathsf{R}' \not\simeq 1$ in WkWd2 and WkWd3 implies that in those rules $\mathsf{S} \not\simeq 1$. We denote this set of transformation rules with $\mathfrak{T}$.

Theorem 4 (Soundness of Unification Rules)**.** *The rules of $\mathfrak{T}$ are sound.*

To solve a unification problem Γ, we create the initial pair $\Gamma; \varepsilon$ and first apply the projection rule to it in all possible ways. From each obtained problem we select an equation and apply the other rules exhaustively to that selected equation, developing the search tree in a breadth-first way. If no rule applies, the problem is transformed to $\perp$. The obtained procedure is denoted by $\mathfrak{P}(\Gamma)$. Branches in the search tree form *derivations*. The derivations that end with $\perp$ are *failing derivations*. The derivations that end with $\emptyset; \varphi$ are *successful derivations*. The set of all φ's at the end of successful derivations of $\mathfrak{P}(\Gamma)$ is called the *computed substitution set* of $\mathfrak{P}(\Gamma)$ and is denoted by $comp(\mathfrak{P}(\Gamma))$. From Theorem 4 by induction on the length of derivations one can prove that every $\varphi \in comp(\mathfrak{P}(\Gamma))$ is a unifier of Γ.

One can observe that under this control, variables are replaced with ϵ only at the projection phase. In particular, no variable introduced in intermediate stages gets eliminated with ϵ or replaced by a variable whose sort is 1.

Theorem 5 (Completeness of the Unification Procedure)**.** *Let Γ be a REOSU problem with a unifier ϑ. Then there exists $\sigma \in comp(\mathfrak{P}(\Gamma))$ such that $\sigma \leq_{var(\Gamma)} \vartheta$.*

Note that the set $comp(\mathfrak{P}(\Gamma))$, in general, is not minimal.[3]

By restricting sorts or occurrences of variables, various terminating fragments of REOSU can be obtained. We mention only four of them here:

- If sorts of all variables in Γ are star-free, then Γ is finitary. To show this, we first transform Γ into Γ', replacing each occurrence of a variable $x : \mathsf{R}_1.\mathsf{R}_2$ in Γ by a sequence of two fresh variables $x_1 : \mathsf{R}_1$ and $x_2 : \mathsf{R}_2$. Then, for each $y : \mathsf{R}_1{+}\mathsf{R}_2$ in Γ', we obtain a new problem Γ'_1 by replacing each occurrence of y by a fresh variable $y_1 : \mathsf{R}_1$, and a a new problem Γ'_2 replacing each occurrence of y by a fresh variable $y_2 : \mathsf{R}_1$. Applying these transformations on each of the obtained problems iteratively, we reach a finite set of order-sorted unification problems, where each variable is of a basic sort. Since the set of basic sorts is finite, such problems are finitary [Wal88]. Γ is solvable if and only if at least one of the obtained problems is solvable. The transformation establishes a one-to-one correspondence between the unifiers of obtained problems and the unifiers of Γ, which implies that Γ is finitary.
- If variables whose sort contains the star occur in the last argument position. This is a pretty useful terminating fragment. One can formulate more optimized transformation rules for it and show termination based on the ideas of a similar fragment in sequence unification [Kut07].

[3]However, if in the rules WkE1 and WkE2 the substitution ω is selected from a minimal subset of the corresponding weakening set, one can show that $comp(\mathfrak{P}(\Gamma))$ is almost minimal. (Almost minimality is defined in [Kut07]).

- The previous fragment can be extended to another terminating fragment, called postfix-closed, where each occurrence of the same star-sorted variable is followed by the same sequence everywhere, like, e.g., in the problem $\{f(a, f(y, b), x, y, b) \doteq f(z, x, y, b)\}$, where the sorts of the variables x and y contain the star.
- If one side of each equation in Γ is ground, then Γ is finitary. These are REOS matching problems. For them, termination of $\mathfrak{P}(\Gamma)$ can be proved based on the ideas of termination proof for sequence matching in [Kut07]. Note that for REOS matching there is no need to invoke the weakening algorithm.

Now we demonstrate on an example how the unification procedure $\mathfrak{P}$ works:

Example 4. Let $\{f(x, y, z) \doteq f(f(x), g(u), a, b)\}$ be a REOSU problem, where the basic sorts are s, r, and q, ordered as $\mathsf{s} \prec \mathsf{q}$, $\mathsf{r} \prec \mathsf{q}$, and the symbols have the following sorts:

$$x, z : \mathsf{s}^* \qquad f : \mathsf{q}^* \to \mathsf{r}$$
$$y, u : \mathsf{q} \qquad g : \mathsf{q} \to \mathsf{q}$$
$$a, b : \mathsf{s} \qquad g : \mathsf{s} + \mathsf{r} \to \mathsf{s}.$$

That means, g is overloaded. We show a successful derivation for this problem. The first two steps are decomposition and projection:

$$\{f(x, y, z) \doteq f(f(x), g(u), a, b)\}; \varepsilon \Longrightarrow_{\mathsf{D}}$$
$$\{(x, y, z) \doteq (f(x), g(u), a, b)\}; \varepsilon \Longrightarrow_{\mathsf{P}}$$
$$\{(y, z) \doteq (f(\epsilon), g(u), a, b)\}; \{x \mapsto \epsilon\}$$

The weakening pair $f(\epsilon) \rightsquigarrow \mathsf{q}$ has ε as a weakening substitution. Hence, we can make the next step with the $\mathsf{WkE1}$ rule:

$$\{(y, z) \doteq (f(\epsilon), g(u), a, b)\}; \{x \mapsto \epsilon\} \Longrightarrow_{\mathsf{WkE1}}$$
$$\{z \doteq (g(u), a, b)\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon)\}$$

Now, $(\mathsf{s}, \mathsf{s}^*) \in \mathit{lf}(\mathit{lsort}(z))$. The least sort of $g(u)$ is $\mathsf{q} \not\preceq \mathsf{s}$. However, we can weaken $g(u)$ towards s: The weakening pair $g(u) \rightsquigarrow \mathsf{s}$ has a solution $\{u \mapsto v\}$, where $v \in \mathcal{V}_{\mathsf{s}+\mathsf{r}}$ is a fresh variable. We perform the $\mathsf{WkWd1}$ step, introducing a fresh variable $z_1 \in \mathcal{V}_{\mathsf{s}^*}$:

$$\{z \doteq (g(u), a, b)\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon)\} \Longrightarrow_{\mathsf{WkWd1}}$$
$$\{z_1 \doteq (a, b)\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon), u \mapsto v, z \mapsto (g(v), z_1)\}$$

The next step is again $\mathsf{WkWd1}$. To make it, we take a weakening substitution ε for $a \rightsquigarrow \mathsf{s}^*$, a fresh variable $z_2 = \mathcal{V}_{\mathsf{s}^*}$ and proceed:

$$\{z_1 \doteq (a, b)\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon), u \mapsto v, z \mapsto (g(v), z_1)\} \Longrightarrow_{\mathsf{WkWd1}}$$
$$\{z_2 \doteq b\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon), u \mapsto v, z \mapsto (g(v), a, z_2), z_1 \mapsto (a, z_2)\}$$

The last two steps in the derivation are $\mathsf{WkE1}$ and T. $\mathsf{WkE1}$ uses the weakening substitution ε for $b \rightsquigarrow \mathsf{s}^*$:

$$\{z_2 \doteq b\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon), u \mapsto v, z \mapsto (g(v), a, z_2), z_1 \mapsto (a, z_2)\} \Longrightarrow_{\mathsf{WkE1}}$$
$$\{\epsilon \doteq \epsilon\}; \{x \mapsto \epsilon, y \mapsto f(\epsilon), u \mapsto v, z \mapsto (g(v), a, b), z_1 \mapsto (a, b), z_2 \mapsto b\} \Longrightarrow_{\mathsf{T}}$$
$$\emptyset; \{x \mapsto \epsilon, y \mapsto f(\epsilon), u \mapsto v, z \mapsto (g(v), a, b), z_1 \mapsto (a, b), z_2 \mapsto b\}.$$

Finally, restricting the computed substitution to the variables of the original problem $\{f(x,y,z) \doteq f(f(x),g(u),a,b)\}$, we obtain its unifier $\{x \mapsto \epsilon,\ y \mapsto f(\epsilon),\ u \mapsto v,\ z \mapsto (g(v),a,b)\}$.

Decidability

To show decidability, we define a translation from REOSU problems into word equations with regular constraints. The idea is similar to the one of [LV01], used to translate context equations into traversal equations, or of [KLV09], used to translate left-hole context equations into word equations with regular constraints.

For each basic sort we assume at least one constant of that sort and proceed as follows:

- First, we show that each solvable REOSU problem Γ has a unifier σ with the property $\mathit{depth}(\sigma) \leq \mathit{size}(\Gamma)$, where $\mathit{size}(\Gamma)$ is the number of alphabet symbols in Γ.
- Next, we transform a REOSU problem Γ into a WU problem with regular constraints by a transformation that preserves solvability in both directions. The transformation uses the minimal unifier depth bound when translating sort information. Since WURC is decidable, we get decidability of REOSU.

We now elaborate on these items. We can assume without loss of generality that we are looking for the unifiers that do not map any variable to ϵ (nonerasing unifiers).

Unifier depth bound. Let ϑ be a depth-minimal nonerasing unifier of Γ with the domain $\mathit{dom}(\vartheta) \subseteq \mathit{var}(\Gamma)$ and let ρ be a grounding substitution for $\Gamma\vartheta$, mapping each variable in $\Gamma\vartheta$ to a sequence of constants of appropriate sort. We denote $\vartheta\rho$ by σ. Then for each $x \in \mathit{var}(\Gamma)$, $x\sigma$ consists of terms of the form $t\sigma$, where t is either a subterm of Γ, or a constant, or is obtained from a subterm of Γ by replacing variables with sequences of constants. Since there are $\mathit{size}(\Gamma)$ subterms in Γ and we can not repeat application of a subterm on itself, $\mathit{depth}(t\sigma) \leq \mathit{size}(\Gamma)$. Therefore, $\mathit{depth}(x\sigma) \leq \mathit{size}(\Gamma)$ for all $x \in \mathit{dom}(\sigma)$ which implies $\mathit{depth}(\sigma) \leq \mathit{size}(\Gamma)$.

Translation into a WURC problem. Let Γ be a REOSU problem. For the translation, we restrict ourselves to the function symbols occurring in Γ and, additionally, one constant for each basic sort, if Γ does not contain a constant of that sort. This alphabet is finite. We denote it by $\mathcal{F}_\Gamma$.

First, we ignore the sort information and define a transformation Tr from term sequences into words as follows:

$$\begin{aligned} \mathit{Tr}(x) &= x \\ \mathit{Tr}(f(\tilde{t})) &= f\,\mathit{Tr}(\tilde{t})f \\ \mathit{Tr}(\epsilon) &= \epsilon \\ \mathit{Tr}(t_1,\ldots,t_n) &= \mathit{Tr}(t_1)\#\cdots\#\mathit{Tr}(t_n),\ n>1 \end{aligned}$$

where # is just a letter that does not occur in $\mathcal{F}_\Gamma$. A mapping σ from variables to term sequences is translated into a substitution for words $\mathit{Tr}(\sigma)$ defined as $x\,\mathit{Tr}(\sigma) = \mathit{Tr}(x\sigma)$ for each x. Tr is an injective function. Its inverse is denoted by Tr^{-1}.

Example 5. Let $\Gamma = \{f(x,y) \doteq f(f(y,a),b,c)\}$ with $\mathsf{s} \preceq \mathsf{r}$, $x:\mathsf{s}$, $y:\mathsf{r}^*$, $f:\mathsf{r}^* \to \mathsf{s}$, $a:\mathsf{s}$ and $b,c:\mathsf{r}$. Then Γ has a solution $\sigma = \{x \mapsto f(b,c,a), y \mapsto (b,c)\}$. On the other hand, $\mathit{Tr}(\Gamma) = \{fx\#yf \doteq ffy\#aaf\#bb\#ccf\}$ is a word unification problem with the nonerasing

solutions $\varphi_1 = \{x \mapsto fbb\#cc\#aaf, y \mapsto bb\#cc\}$, $\varphi_2 = \{x \to fcc\#aaf\#bb, y \mapsto cc\}$, $\varphi_3 = \{x \mapsto faaf\#bbf\#cc, y \mapsto aaf\#bbf\#cc\}$. It is easy to see that $\varphi_1 = Tr(\sigma)$, but φ_2 and φ_3 are extra substitutions introduced by the transformation. However, they are of different nature: $Tr^{-1}(\varphi_2)$ exists and it is a mapping $\{x \mapsto (f(c,a),b), y \mapsto c\}$, but it is not a substitution because it is not well-sorted. $Tr^{-1}(\varphi_3)$ does not exist (which indicates that Tr is not surjective).

Lemma 5. *If σ is a substitution and $\tilde{t}$ is a sequence of REOS terms, then $Tr(\tilde{t})Tr(\sigma) = Tr(\tilde{t}\sigma)$.*

This lemma implies that if a REOSU Γ is solvable, then $Tr(\Gamma)$ is solvable. The converse, in general, is not true, because the transformation introduces extra solutions. However, translating sort information and considering word equations with regular constraints prevents extra solutions to appear and we get solvability preservation in both directions, as we will see below.

We start with translating sort information: For each $x \in var(\Gamma)$, we transform $x : \mathsf{R}$ into a membership constraint $x \in Tr(\mathsf{R}, \Gamma)$, where $Tr(\mathsf{R}, \Gamma)$ is defined as the set

$$Tr(\mathsf{R},\Gamma) = \{\, Tr(\tilde{t}) \mid \text{the terms in } \tilde{t} \text{ are from } \mathcal{T}(\mathcal{F}_\Gamma), \; lsort(\tilde{t}) \preceq \mathsf{R} \text{ and } depth(\tilde{t}) \leq size(\Gamma)\}.$$

That is, we translate only those $\tilde{t}$'s whose minimal sort does not exceed R and whose depth is bounded by $size(\Gamma)$.

We show now that $Tr(\mathsf{R}, \Gamma)$ is a regular word language. First, we introduce a notation for regular word languages: We write $L_1 ._\# L_2$ for the language $\{w_1\#w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. $L^{0\#} = \{\epsilon\}$, $L^{1\#} = L$, $L^{n\#} = L ._\# L^{(n-1)\#}$ and $L^{*\#} = \cup_{n=0}^{\infty} L^{n\#}$.

For each $\mathcal{R}$, the language $Tr(\mathsf{R}, \Gamma)$ is constructed level by level, first for the term sequences of depth 1, then for depth 2, and so on, until the depth bound $depth(\Gamma)$:

- Depth 1:

$$\begin{aligned}
Tr_1(\mathsf{s},\Gamma) &= \{aa \mid a \in \mathcal{F}_\Gamma, a : \mathsf{s}', \mathsf{s}' \preceq \mathsf{s}\} \text{ (This set is finite.)}\\
Tr_1(1,\Gamma) &= \{\epsilon\}\\
Tr_1(\mathsf{R}_1 + \mathsf{R}_2,\Gamma) &= Tr_1(\mathsf{R}_1,\Gamma) \cup Tr_1(\mathsf{R}_2,\Gamma)\\
Tr_1(\mathsf{R}_1.\mathsf{R}_2,\Gamma) &= Tr_1(\mathsf{R}_1,\Gamma) ._\# Tr_1(\mathsf{R}_2,\Gamma)\\
Tr_1(\mathsf{R}^*,\Gamma) &= Tr_1(\mathsf{R},\Gamma)^{*\#}
\end{aligned}$$

- Depth $n > 1$:

$$\begin{aligned}
Tr_n(\mathsf{s},\Gamma) &= Tr_{n-1}(\mathsf{s},\Gamma) \cup \{fwf \mid f \in \mathcal{F}_\Gamma, f : \mathsf{R} \to \mathsf{s}',\\
&\qquad w \in Tr_{n-1}(\mathsf{R}',\Gamma), \mathsf{R}' \preceq \mathsf{R}, \mathsf{s}' \preceq \mathsf{s}\}\\
Tr_n(1,\Gamma) &= \{\epsilon\}\\
Tr_n(\mathsf{R}_1 + \mathsf{R}_2,\Gamma) &= Tr_n(\mathsf{R}_1,\Gamma) \cup Tr_n(\mathsf{R}_2,\Gamma)\\
Tr_n(\mathsf{R}_1.\mathsf{R}_2,\Gamma) &= Tr_n(\mathsf{R}_1,\Gamma) ._\# Tr_n(\mathsf{R}_2,\Gamma)\\
Tr_n(\mathsf{R}^*,\Gamma) &= Tr_n(\mathsf{R},\Gamma)^{*\#}
\end{aligned}$$

It shows that $Tr_n(\mathsf{R}, \Gamma)$ is regular for each n. From this construction it follows that $Tr(\mathsf{R}, \Gamma) = Tr_{size(\Gamma)}(\mathsf{R}, \Gamma)$ and, hence, $Tr(\mathsf{R}, \Gamma)$ is regular.

Example 6. Consider again Γ and the sort information from Example 5. Now it gets translated into a WURC problem $\Delta = \{fx\#yf \doteq ffy\#aaf\#bb\#ccf,\ x \in \mathit{Tr}(\mathsf{s},\Gamma), y \in \mathit{Tr}(\mathsf{r}^*,\Gamma)\}$. $\mathit{Tr}(\mathsf{s},\Gamma)$ contains (among others) $fbb\#cc\#aaf$, but neither $fcc\#aaf\#bb$ nor $faaf\#bbf\#cc$. $\mathit{Tr}(\mathsf{r}^*,\Gamma)$ contains (among others) $bb\#cc$. Hence, φ_1 from Example 5 is a solution of Δ, but φ_2 and φ_3 are not.

Finally, we have the theorem:

Theorem 6. *Let* $\Gamma = \{\tilde{s}_1 \doteq \tilde{t}_1, \dots, \tilde{s}_n \doteq \tilde{t}_n\}$ *be a REOSU problem with* $\mathit{var}(\Gamma) = \{x_1, \dots, x_m\}$ *such that* $x_i : \mathsf{R}_i$ *for each* $1 \leq i \leq m$. *Let* $\Delta = \{\mathit{Tr}(\tilde{s}_1) \doteq \mathit{Tr}(\tilde{t}_1), \dots, \mathit{Tr}(\tilde{s}_n) \doteq \mathit{Tr}(\tilde{t}_n), x_1 \in \mathit{Tr}(\mathsf{R}_1,\Gamma), \dots, x_m \in \mathit{Tr}(\mathsf{R}_m,\Gamma)\}$ *be a word unification problem with regular constraints, obtained by translating* Γ. *Then* Γ *is solvable iff* Δ *is solvable.*

Hence, the problem of deciding solvability of REOSU has been (polynomially) reduced to the problem of deciding solvability of WURC. Since the latter is decidable, we conclude with the following result:

Theorem 7 (Decidability)**.** *Solvability of REOSU is decidable.*

4. Conclusion

We studied unification in order-sorted theories with regular expression sorts. We showed how it generalizes some known unification problems, proved its decidability and gave a complete unification procedure. A regular expression order-sorted signature can be viewed as a bottom-up finite hedge automaton. Such automata are considered to be a suitable framework for manipulating XML data. Since our language can model, to some extent, DTD and XML Schema, one can see a possible application (perhaps of its fragments) in the area related to XML processing.

Acknowledgments

This research has been partially supported by the EC FP6 Programme for Integrated Infrastructures Initiatives under the project SCIEnce—Symbolic Computation Infrastructure for Europe (Contract No. 026133) and by JSPS Grant-in-Aid no. 20500025 for Scientific Research (C).

References

[AM95] V. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.

[Ant95] V. Antimirov. Rewriting regular inequalities (extended abstract). In H. Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium FCT'95*, volume 965 of *LNCS*, pages 116–125. Springer, 1995.

[Ant96] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[Bou92] A. Boudet. Unification in order-sorted algebras with overloading. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction, CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1992.

[BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[CDG+] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available from: `http://www.grappa.univ-lille3.fr/tata`, version from October 12, 2007.

[Com89] H. Comon. Inductive proofs by specification transformation. In N. Dershowitz, editor, *Proc. 3rd International Conference on Rewriting Techniques and Applications, RTA'89*, volume 355 of *LNCS*, pages 76–91. Springer, 1989.

[Con71] J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[GM92] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[HM08] J. Hendrix and J. Meseguer. Order-sorted unification revisited. In G. Kniesel and J. Sousa Pinto, editors, *Pre-proceedings of the 9th International Workshop on Rule-Based Programming, RULE'08*, pages 16–29, 2008.

[HP03] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. *J. Functional Programming*, 13(6):961–1004, 2003.

[Kir88] C. Kirchner. Order-sorted equational unification. Presented at the fifth International Conference on Logic Programming (Seattle, USA), 1988. Also as rapport de recherche INRIA 954, December 1988.

[KLV09] T. Kutsia, J. Levy, and M. Villaret. On the relation between context and sequence unification. *J. Symbolic Computation*, 45(1):74–95, 2009.

[Kut02] T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proc. of Joint AISC'2002 – Calculemus'2002 Conference*, volume 2385 of *LNAI*, pages 290–304. Springer, 2002.

[Kut07] T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symbolic Computation*, 42(3):352–388, 2007.

[LV01] J. Levy and M. Villaret. Context unification and traversal equations. In A. Middeldorp, editor, *Proc. of the 12th International Conference on Rewriting Techniques and Applications, RTA'01*, volume 2041 of *LNCS*, pages 169–184. Springer, 2001.

[MGS89] J. Meseguer, J. A. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8(4):383–413, 1989.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Sch90] K. U. Schulz. Makanin's algorithm for word equations – two improvements and a generalization. In K. Schulz, editor, *Word Equations and Related Topics*, number 572 in LNCS, pages 85–150. Springer, 1990.

[SNGM89] G. Smolka, W. Nutt, J. A. Goguen, and J. Meseguer. Order-sorted equational computation. In M. Nivat and H. Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 297–367. Academic Press, 1989.

[SS89] M. Schmidt-Schauß. *Computational Aspects of an Order-sorted Logic with Term Declarations*. Number 395 in Lecture Notes in Computer Science. Springer, 1989.

[Sul09] M. Sulzmann. regexpr-symbolic: Regular expressions via symbolic manipulation. `http://hackage.haskell.org/package/regexpr-symbolic`, 2009.

[Wal88] Ch. Walther. Many-sorted unification. *J. ACM*, 35(1):1–17, 1988.

[Wei96] Ch. Weidenbach. Unification in sort theories and its applications. *Annals of Mathematics and Artificial Intelligence*, 18(2):261–293, 1996.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 209-226
http://rewriting.loria.fr/rta/

AN EFFICIENT NOMINAL UNIFICATION ALGORITHM

JORDI LEVY [1] AND MATEU VILLARET [2]

[1] Artificial Intelligence Research Institute (IIIA),
Spanish Council for Scientific Research (CSIC), Barcelona, Spain.
E-mail address: levy@iiia.csic.es
URL: http://www.iiia.csic.es/~levy

[2] Departament d'Informàtica i Matemàtica Aplicada (IMA),
Universitat de Girona (UdG), Girona, Spain.
E-mail address: villaret@ima.udg.edu
URL: http://ima.udg.edu/~villaret

Abstract. Nominal Unification is an extension of first-order unification where terms can contain binders and unification is performed modulo α-equivalence. Here we prove that the existence of nominal unifiers can be decided in quadratic time. First, we linearly-reduce nominal unification problems to a sequence of freshness and equalities between atoms, modulo a permutation, using ideas as Paterson and Wegman for first-order unification. Second, we prove that solvability of these reduced problems may be checked in quadratic time. Finally, we point out how using ideas of Brown and Tarjan for unbalanced merging, we could solve these reduced problems more efficiently.

1. Introduction

Nominal techniques introduce mechanisms for renaming via name-swapping, for name-binding, and for freshness of names. They were introduced at the beginning of this decade by Gabbay and Pitts [Pit01, Gab01, Pit03]. These first works have inspired a sequel of papers where bindings and freshness are introduced in other topics, like nominal algebra [Gab06, Gab07, Gab09], equational logic [Clo07], rewriting [Fer05, Fer07], unification [Urb03, Urb04], and Prolog [Che04, Urb05].

In this paper we study the complexity of *Nominal Unification* [Urb03, Urb04], an extension of first-order unification where terms can contain binders and unification is performed modulo α-equivalence. Moreover, (first-order) variables (*unknowns*) are allowed to "capture" bound variables (*atoms*) contrarily to unification in λ-calculus. In [Urb03, Urb04] it is described a sound and complete, but inefficient (exponential), algorithm for nominal unification. Later this algorithm was extended to deal with the new-quantifier and locality in [Fer05]. In [Cal07] there is a description of a direct but exponential implementation in

Key words and phrases: Nominal Logic, Unification.

This research has been partially founded by the CICYT research projects Mulog2 (TIN2007-68005-C04) and SuRoS (TIN2008-04547).

© J. Levy and M. Villaret
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.209

Maude, and a polynomial implementation in OCAML based on termgraphs. In [Cal08], it is described a polynomial algorithm for nominal unification. In [Lev08] it is proved that the problem can be solved in quadratic time by quadratic reduction to Higher-Order Pattern Unification, that is claimed to be linear [Qia96]. Therefore, the present algorithm does not improve the complexity bounds already known. However, it has to be noticed that in this paper we describe a practical implementation, and that it is really difficult to obtain a practical algorithm from the proof described in [Qia96]. In [Cal10] there is a quadratic algorithm for nominal unification, independently found by Calvès, and also based on Paterson and Wegman's first-order unification algorithm. Other extensions of nominal unification have been studied in [Che05, Dow09, Dow10].

This paper proceeds as follows. In Section 2 we describe nominal logic and the nominal unification algorithm of [Urb03, Urb04]. In Section 3 we prove that freshness equations and suspensions are mere syntactic sugar. We can translate them in terms of basic nominal equations, with a linear increasing in the size of the problem. In Section 4 we describe the Paterson-Wegman linear algorithm for First-Order Unification [Pat78] and some preliminary ideas of how we plan to adapt this algorithm to nominal unification. In Section 5, we introduce *replacings* as $L = (a_1 \leftarrow b_1) \cdots (a_n \leftarrow b_n)$. We say that t and u are equivalent modulo L, written $t =_L u$, if $a_n. \cdots a_1.t \approx b_n. \cdots b_1.u$. In some cases we need to compute some kind of composition of replacings. This leads us to the introduction of *generalized replacings* in Section 6. The adaptation of Paterson-Wegman's algorithm is described in Section 7. It allows us to translate a nominal unification problem into a set of replacing equations in linear time. Section 8 is devoted to the verification of these replacing equations. There we prove that it can be done in quadratic time. Finally, in Section 9 we discuss on the possibility of improving this bound and do this verification in quasi-linear time.

2. Preliminaries

Nominal terms contain *variables* and *atoms*. Only variables may be instantiated, and only atoms may be bound. They roughly correspond to the higher-order notions of free and bound variables, respectively, but are considered as completely different entities. Therefore, contrarily to the higher-order perspective, in nominal terms it makes no sense the distinction between free and bound variables depending on the existence of a binder above them.

Nominal terms[1] (typically $t, u, \dots$) are given by the grammar:

$$t ::= \langle t_1, t_2 \rangle \mid f(t_1, \dots, t_n) \mid a \mid a.t \mid \pi\, X$$

where f is a function symbol, a is an atom, π is a permutation (finite list of swappings), and X is a variable.

A *swapping* $(a\, b)$ is a pair of atoms of the same sort. The effect of a swapping over an atom is defined by $(a\, b)\, a = b$ and $(a\, b)\, b = a$ and $(a\, b)\, c = c$, when $c \neq a, b$. For the rest of terms the extension is straightforward, in particular, $(a\, b)\, (c.t) = \big((a\, b)\, c\big).\big((a\, b)\, t\big)$. A *permutation* is a (possibly empty) sequence of swappings. *Suspensions* are uses of variables with a permutation of atoms waiting to be applied once the variable is instantiated.

Substitutions are sort-respecting functions and behave like in first-order logic, hence allowing atom capture, for instance $[X \mapsto a]a.X = a.a$.

[1] For simplicity, we do not consider the *unit value* nor the *pairing*. Instead of them we consider n-ary function symbols.

A *freshness environment* (typically ∇) is a list of *freshness constraints* $a \# X$ stating that the instantiation of X cannot contain free occurrences of a.

The notion of nominal term α-*equivalence*, noted $\approx$, is defined by means of the following theory:

$$\frac{\nabla \vdash t_1 \approx u_1 \cdots \nabla \vdash t_n \approx u_n}{\nabla \vdash f(t_1,\dots,t_n) \approx f\,(u_1,\dots,u_n)}\ (\approx\text{-function}) \qquad \frac{}{\nabla \vdash a \approx a}\ (\approx\text{-atom})$$

$$\frac{a \neq a' \quad \nabla \vdash t \approx (a\,a')\,t' \quad \nabla \vdash a\#t'}{\nabla \vdash a.t \approx a'.t'}\ (\approx\text{-abst-2}) \qquad \frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'}\ (\approx\text{-abst-1})$$

$$\frac{(a\#X) \in \nabla \text{ for all } a \text{ such that } \pi\, a \neq \pi'\, a}{\nabla \vdash \pi\, X \approx \pi'\, X}\ (\approx\text{-susp.})$$

where the *freshness* predicate $\#$ is defined by:

$$\frac{\nabla \vdash a\#t_1 \cdots \nabla \vdash a\#t_n}{\nabla \vdash a\#f(t_1,\dots,t_n)}\ (\#\text{-function}) \qquad \frac{a \neq a'}{\nabla \vdash a\#a'}\ (\#\text{-atom})$$

$$\frac{}{\nabla \vdash a\#a.t}\ (\#\text{-abst-1}) \qquad \frac{a \neq a' \quad \nabla \vdash a\#t}{\nabla \vdash a\#a'.t}\ (\#\text{-abst-2}) \qquad \frac{(\pi^{-1}\, a\,\#X) \in \nabla}{\nabla \vdash a\#\pi\, X}\ (\#\text{-susp.})$$

Their intended meanings are: $\nabla \vdash a \# t$ holds if, for every substitution σ respecting the freshness environment ∇ (i.e. avoiding the atom captures forbidden by ∇), a is not free in $\sigma(t)$; $\nabla \vdash t \approx u$ holds if, for every substitution σ respecting the freshness environment ∇, t and u are α-convertible.

A *nominal unification problem* (typically P) is a set of equations of the form $t \stackrel{?}{\approx} u$ or $a \#^{?} t$, *equational problems* and *freshness problems* respectively. A *solution* of a nominal problem is given by a substitution σ and a freshness environment ∇. Formally, the pair $\langle \nabla, \sigma \rangle$ solves P if, $\nabla \vdash a \# \sigma(t)$, for freshness problems $a \#^{?} t \in P$, and $\nabla \vdash \sigma(t) \approx \sigma(u)$, for equational problems $t \stackrel{?}{\approx} u \in P$.

Example 2.1. The solutions of the equation $a.X \stackrel{?}{\approx} b.X$ can not instantiate X with terms containing free occurrences of the atoms a and b, for instance if we apply the substitution $[X \mapsto a]$ to both sides of the equation we get $[X \mapsto a]a.X = a.a$ for the left hand side and $[X \mapsto a]b.X = b.a$ for the right hand side, and obviously $a.a \not\approx b.a$.

The most general solution of this equation is $\langle \{a\#X, b\#X\}, [\,] \rangle$.

The first linear First-Order Unification algorithm was described by Paterson and Wegman [Pat78]. Here we describe it in terms of transformation rules as it is done by Martelli and Montanari in [Mar82].

Definition 2.2. The Paterson-Wegman can be described by the following two transformation rules.

Simplification:

$$\left.\begin{array}{l} \{X_1, X'_1, \dots\} = f(Y_1,\dots,Y_m) \\ \{X_2, X'_2, \dots\} = f(Z_1,\dots,Z_m) \\ X_1 = X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \{X_1, X'_1, \dots, X_2, X'_2, \dots\} = f(Y_1,\dots,Y_m) \\ Y_1 = Z_1 \\ \dots \\ Y_m = Z_m \end{array}\right.$$

Variable:

$$\left.\begin{array}{l}\{X_1, X_1', \dots\} = t \\ \{X_2, X_2', \dots\} = \emptyset \\ X_1 = X_2\end{array}\right\} \Longrightarrow \{X_1, X_1', \dots, X_2, X_2', \dots\} = t$$

At every transformation, the selected equation $X_1 = X_2$ has to be maximal in the sense that there is no other equation $X_1^m = X_2^m$ and a set of equations of the form $\{X_i^m, \dots\} = f_{m-1}(\dots, X_i^{m-1}, \dots),\ \dots\ , \{X_i^1, \dots\} = f_1(\dots, X_i, \dots)$ for $i = 1$ or $i = 2$.

3. Three Initial Simplifications

In this section we show how we can simplify nominal unification problems getting rid of freshness equations, of suspensions, and flattening all applications and abstractions. We will show that these simplifications only increase the size of the problem linearly. Lemma 3.1 shows us how to encode a freshness equation as an equality equation, and Lemma 3.2, how to encode a suspension also as an equality. Therefore, we can conclude that freshness equations and suspensions are mere syntactic sugar in nominal unification.

Lemma 3.1. *Let $b \neq a$. Then, $P \cup \{a\#t\}$ and $P \cup \{a.b.t \stackrel{?}{\approx} b.b.t\}$ have the same solutions.*

Proof. We first prove that $\langle a\#t, Id\rangle$ is a solution of $\{a.b.t \stackrel{?}{\approx} b.b.t\}$ when $b \neq a$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ t \approx t\end{array} \quad \begin{array}{c}a\#t\\ \vdots\ \text{(lemma 2.7)}\\ b\#(a\,b)\,t\end{array}}{b.t \approx a.(a\,b)\,t}\ (\approx\text{-abst-2}) \qquad \dfrac{a\#t}{a\#b.t}\ (\#\text{-abst-2})}{a.b.t \approx b.b.t}\ (\approx\text{-abst-2})$$

In this proof we prove $t \approx t$ from an empty set of assumptions. We can prove that this is always possible, for any term t, by structural induction on t. We also prove $b\#(a\,b)\,t$ from $a\#t$, using Lemma 2.7 of [Urb04].

Now, since $\nabla' \vdash \sigma(\nabla)$ and $\nabla \vdash t \approx t'$ implies $\nabla' \vdash \sigma(t) \approx \sigma(t')$ (see Lemma 2.14 of [Urb04]), we have that, if $\langle \nabla, \sigma\rangle$ solves $a\#^? t$, then $\langle \nabla, \sigma\rangle$ solves $a.b.t \stackrel{?}{\approx} b.b.t$.

Second, analyzing the previous proof, we see that the inference rules applied in each situation were the only applicable rules. Therefore, any solution $\langle \nabla, \sigma\rangle$ solving $a.b.t \stackrel{?}{\approx} b.b.t$, also solves $a\#^? t$, because any proof of $\sigma(a.b.t) \approx \sigma(b.b.t)$ contains a proof of $a\#\sigma(t)$ as a sub-proof.

From, these two fact we conclude that $a \#^? t$ and $a.b.t \stackrel{?}{\approx} b.b.t$ have the same set of solutions, for any $b \neq a$. Therefore, $\{a \#^? t\} \cup P$ and $\{a.b.t \stackrel{?}{\approx} b.b.t\} \cup P$, also have the same set of solutions, for any nominal unification problem P. From this we conclude that $P \cup \{a\#t\}$ and $P \cup \{a.b.t \stackrel{?}{\approx} b.b.t\}$ have the same set of solutions. ∎

Lemma 3.2. *$P \cup \{t \stackrel{?}{\approx} (a\,b)u\}$ and $P \cup \{a.b.t \stackrel{?}{\approx} b.a.u\}$ have the same solutions.*

Proof. If $a = b$ the proof is obvious. If $a \neq b$, then the proof is similar to proof of Lemma 3.1. In this case, the proof of $a.b.t \stackrel{?}{\approx} b.a.u$ from $t \stackrel{?}{\approx} (a\,b)u$ is as follows:

$$\dfrac{\dfrac{t \approx (a\,b)u}{b.t \approx b.(a\,b)u}\ (\approx\text{-abst-1}) \qquad \dfrac{}{a\#a.u}\ (\#\text{-abst-1})}{a.b.t \approx b.a.u}\ (\approx\text{-abst-2})$$

■

Lemma 3.3. *Let X be a fresh variable not occurring elsewhere. Then,*
$P \cup \{a.t \overset{?}{\approx} u\}$ *and* $P \cup \{a.X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$ *are equivalent*
$P \cup \{f(t_1,\dots,t_n) \overset{?}{\approx} u\}$ *and* $P \cup \{f(t_1,\dots,t_{i-1},X,t_{i+1},t_n) \overset{?}{\approx} u, X \overset{?}{\approx} t_i\}$ *are equivalent,*
$P \cup \{(a\,b)\,t \overset{?}{\approx} u\}$ *and* $P \cup \{(a\,b)\,X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$ *are equivalent,*
$P \cup \{t_1 \overset{?}{\approx} t_2\}$ *and* $P \cup \{X \overset{?}{\approx} t_1, X \overset{?}{\approx} t_2\}$ *are equivalent, and*
$P \cup \{Y_1 \overset{?}{\approx} Y_2\}$ *and* $[Y_1 \mapsto Y_2]P$ *are equivalent.*

Proof. Let us consider the first statement. If $\langle \nabla, \sigma\rangle$ solves $P \cup \{a.t \overset{?}{\approx} u\}$, then it is enough to extend σ with $X \mapsto \sigma(t)$ to get a solution of $P \cup \{a.X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$. In the opposite direction, any solution of $P \cup \{a.X \overset{?}{\approx} u, X \overset{?}{\approx} t\}$ is a solution of $P \cup \{a.t \overset{?}{\approx} u\}$, because, for any three terms t_1, t_2 and t_3, if $a.t_2 \approx t_1$ and $t_2 \approx t_3$, then $a.t_3 \approx t_1$. ■

Notice that the previous lemma does not hold for unification in λ-calculus. For instance, $\{\lambda a.f(a) =^? \lambda b.f(b)\}$ is trivially solvable. However, $\{\lambda a.X =^? \lambda b.b, X =^? a\}$ is unsolvable because, in λ-calculus, we have to avoid variable-capture in substitutions. This fact prevented Qian [Qia96] to apply this simplification in his linear-time algorithm for higher-order pattern unification.

Theorem 3.4. *There exists a linear reduction from Nominal Unification to a simplified version of Nominal Unification where all equations are of the form $X \overset{?}{\approx} a$, $X \overset{?}{\approx} f(Y_1,\dots,Y_n)$ or $X \overset{?}{\approx} a.Y$.*

Proof. We apply four reductions. First, applying Lemma 3.1, we can remove all freshness equations. Second, applying the transformations of Lemma 3.3 widely, replacing the first set of equations by the second whenever t is not a variable (in the first and third rules), or t_i is not a variable (in the second rule), or t_1 and t_2 are not variables (in the forth rule), we can flat all equations. Now, all equations have a variable in one side and a term of the form a, $a.X$, $f(X_1,\dots,X_n)$, or $(a\,b)\,X$ in the other side. In particular, all suspensions will occur in equations of the form $X \overset{?}{\approx} (a\,b)Y$. Applying Lemma 3.2, we can remove all them, translating them into $a.b.X \overset{?}{\approx} b.a.Y$. Forth, all these equations can be translated into $Z_3 \overset{?}{\approx} a.Z_1$, $Z_3 \overset{?}{\approx} b.Z_2$, $Z_1 \overset{?}{\approx} b.X$, $Z_2 \overset{?}{\approx} a.Y$, where Z_1, Z_2 and Z_3 are fresh.

A simple analysis shows that all these transformations are linear. ■

4. A First (Naive) Idea

Considering the similarities between Nominal Unification and FO Unification, a natural way to address the implementation of an efficient nominal unification algorithm is to postpone as much as possible the test of freshness predicates and equality between atoms. We can adapt algorithm of Definition 4.1 as follows. Instead of equations between variables, we use equations between variables affected by a permutation: $X_1 = \pi X_2$. Moreover, these equations are coupled with a set of freshness restrictions with the form of an implication: $a \neq \pi_1 b_1 \wedge \cdots \wedge a \neq \pi_n b_n \Rightarrow a \# \pi_0 X_2$. The application rule is quite similar to the one used in algorithm 4.1, but the abstraction rule involves the extension of the permutation, the addition of a new associated freshness restriction and of additional conditions to the rest of freshness restrictions.

Definition 4.1. Consider the following (sound but incomplete) nominal unification algorithm. Given a set of simplified equations, transform them into a set of multi-equations as follows. First, transform any equation $X \stackrel{?}{=} t$ into a multi-equation $\{X\} = t$, and second, transform any pair of multi-equations $\{X\} = t_1, \{X\} = t_2$ into $\{X\} = t_1, \{X'\} = t_2$, $X = X'$, and add a multi-equation $\{X\} = \emptyset$ for any variable not occurring in the left of any multi-equation, until all variables occur in the left of a multi-equation exactly once. Then, apply the following transformation rules wisely.

Application:

$$\left.\begin{array}{l} \{X_1, S_1\} = f(Y_1, \dots, Y_m) \\ \{X_2, S_2\} = f(Z_1, \dots, Z_m) \\ X_1 = \pi X_2 \\ P_1 \Rightarrow c_1 \# \pi_1 X_2 \\ \cdots \\ P_n \Rightarrow c_n \# \pi_n X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \{X_1, S_1, \pi X_2, \pi S_2\} = f(Y_1, \dots, Y_m) \\ \\ Y_1 = \pi Z_1, \cdots, Y_m = \pi Z_m \\ P_1 \Rightarrow c_1 \# \pi_1 Z_1, \dots, P_1 \Rightarrow c_1 \# \pi_1 Z_m \\ \cdots \\ P_n \Rightarrow c_n \# \pi_n Z_1, \dots, P_n \Rightarrow c_n \# \pi_n Z_m \end{array}\right.$$

Abstraction:

$$\left.\begin{array}{l} \{X_1, S_1\} = a.Y \\ \{X_2, S_2\} = b.Z \\ X_1 = \pi X_2 \\ P_1 \Rightarrow c_1 \# \pi_1 X_2 \\ \cdots \\ P_n \Rightarrow c_n \# \pi_n X_2 \\ \\ \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \{X_1, S_1, \pi X_2, \pi S_2\} = a.Y \\ \\ Y = (a\ \pi b)\pi Z \\ P_1 \wedge c_1 \neq \pi_1\, b \Rightarrow c_1 \# \pi_1 Z \\ \cdots \\ P_n \wedge c_n \neq \pi_1\, b \Rightarrow c_n \# \pi_n Z \\ a \neq \pi b \Rightarrow a \# \pi Z \end{array}\right.$$

Atom:

$$\left.\begin{array}{l} \{X_1, S_1\} = a \\ \{X_2, S_2\} = b \\ X_1 = \pi X_2 \\ P_1 \Rightarrow c_1 \# \pi_1 X_2 \\ \cdots \\ P_n \Rightarrow c_n \# \pi_n X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \{X_1, S_1, \pi X_2, \pi S_2\} = a \\ \\ a = \pi b \\ P_1 \Rightarrow c_1 \neq \pi_1 b \\ \cdots \\ P_n \Rightarrow c_n \neq \pi_n b \end{array}\right.$$

Notice that the algorithm previously described is incomplete. For instance, the variable X_1 in $\{X_1, S_1\} = f(Y_1, \dots, Y_m)$ could be already affected by a permutation, which makes the rule inapplicable. However, these rules allow us to solve the following example:

Example 4.2. The Nominal unification problem $a_3.a_2.a_1.f(c_1, c_2) \stackrel{?}{\approx} b_3.b_2.b_1.f(d_1, d_2)$ is transformed by the naive algorithm into the following set of conditional equalities and inequalities.

$$\begin{array}{l} c_1 = (a_1\ (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1)(a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_1 \\ c_2 = (a_1\ (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1)(a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_2 \\ a_3 \neq b_3\ \wedge\ a_3 \neq b_2\ \wedge\ a_3 \neq b_1\ \Rightarrow\ a_3 \neq d_1 \\ a_3 \neq b_3\ \wedge\ a_3 \neq b_2\ \wedge\ a_3 \neq b_1\ \Rightarrow\ a_3 \neq d_2 \\ a_2 \neq (a_3\ b_3) b_2\ \wedge\ a_2 \neq (a_3\ b_3) b_1\ \Rightarrow\ a_2 \neq (a_3\ b_3) d_1 \\ a_2 \neq (a_3\ b_3) b_2\ \wedge\ a_2 \neq (a_3\ b_3) b_1\ \Rightarrow\ a_2 \neq (a_3\ b_3) d_2 \\ a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1\ \Rightarrow\ a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_1 \\ a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) b_1\ \Rightarrow\ a_1 \neq (a_2\ (a_3 b_3) b_2)(a_3\ b_3) d_2 \end{array}$$

It is easy to see that a generalization of this simple problem to

$$a_n.\dots.a_1.f(c_1,\dots,c_m) \stackrel{?}{\approx} b_n.\dots.b_1.f(d_1,\dots,d_m)$$

would result in a set of inequalities of size $\mathcal{O}(n\,m)$. The number of comparisons of atoms that have to be checked in order to compute the result of applying the permutation and check the equalities is also $\mathcal{O}(n\,m)$.

5. Simple Replacings

In this section we introduce a new concept, similar to the idea of substitution and of swapping, but with some differences. Thus, we have preferred to call it with the new name *replacings*.

Definition 5.1. A *replacing* is a (possibly empty) list of pairs of atoms $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$.

Given two terms t and u and a replacing $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$, we say that t and u are equivalent modulo L, noted $t =_L u$, if $a_n.\cdots a_1.t \approx b_n.\cdots b_1.u$.

Any replacing may be associated with a permutation of atoms, defined as follows. This definition and the following lemma, helps us to see replacings as permutations, plus a set of associated freshness equations. The example bellow also shows that the associated permutation is not enough to characterize a replacing.

Definition 5.2. Given a replacing L, we define its associated permutation Π_L inductively as follows

(1) $\Pi_{[\,]} = [\,]$, being $[\,]$ the empty list, and empty sequence of swappings.
(2) $\Pi_{(a\leftarrow b)L} = (a\ \Pi_L b)\Pi_L$

Lemma 5.3. *Given a replacing $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ and two terms t and u, $t =_L u$ holds, iff*

(1) $t \approx \Pi_L u$, *and*
(2) *for any $i = 1,\dots,n$, if $a_i \neq \Pi_{(a_{i+1}\leftarrow b_{i+1})\dots(a_n\leftarrow b_n)} b_j$ for all $j = i,\dots,1$, then $a_i \# \Pi_{(a_{i+1}\leftarrow b_{i+1})\dots(a_n\leftarrow b_n)} u$.*

Example 5.4. Notice that the permutation Π_L does not characterize the replacing L. For instance, we have

$$\Pi_{(a\leftarrow b)} = \Pi_{(b\leftarrow a)} = \Pi_{(b\leftarrow a)(a\leftarrow b)} = \Pi_{(a\leftarrow b)(b\leftarrow a)} = \Pi_{(a\leftarrow b)(a\leftarrow b)} = (a\ b) = (b\ a)$$

However, assuming $a \neq b$, we have

$$\begin{array}{rcccl} t =_{(a\leftarrow b)} u & \Leftrightarrow & t =_{(a\leftarrow b)(a\leftarrow b)} u & \Leftrightarrow & t = (a\ b)u \ \wedge\ a\#u \\ t =_{(b\leftarrow a)} u & \Leftrightarrow & t =_{(b\leftarrow a)(b\leftarrow a)} u & \Leftrightarrow & t = (a\ b)u \ \wedge\ b\#u \\ t =_{(b\leftarrow a)(a\leftarrow b)} u & \Leftrightarrow & t =_{(a\leftarrow b)(b\leftarrow a)} u & \Leftrightarrow & t = (a\ b)u \end{array}$$

If for any pair of term we have $t =_L u \Leftrightarrow t =_{L'} u$, then this will be also true for any pair of atoms, and we will have $\Pi_L = \Pi_{L'}$. This motivates the following definition.

Definition 5.5. We say that two replacings L and L' are equivalent if, for any pair of terms t and u, we have $t =_L u$ iff $t =_{L'} u$

Lemma 5.6. $t =_{(a_1\leftarrow b_1)\cdots(a_n\leftarrow b_n)} u$ *iff* $u =_{(b_1\leftarrow a_1)\cdots(b_n\leftarrow a_n)} t$.

The following lemma describes a method to check if $c =_L d$ in time $\mathcal{O}(|L|)$.

Lemma 5.7. *Given two atoms c and d and a replacing $(a \leftarrow b)L$:*

$$c =_{(a\leftarrow b)L} d \quad \textit{iff} \quad \begin{array}{l} c = a \text{ and } b = d, \text{ or} \\ c \neq a,\ b \neq d \text{ and } c =_L d. \end{array}$$

Next, we will describe a normalization procedure of replacings. We say that a replacing $(a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ is normalized if $a_1, \dots, a_n$ is a list of pairwise distinct atoms, and $b_1, \dots, b_n$ too. Lemma 5.8 states that, any normalized replacing may be characterized by a *set*, instead of a *list*), of pairs of atoms. Lemma 5.9 shows how we can remove duplicated pairs and normalized replacings, on the expenses of adding freshness equations.

When atoms are not repeated in a replacing, then they are basically[2] a permutation, as the following lemma states.

Lemma 5.8. *If $L = (a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ is a normalized replacing, i.e. a replacing where $a_1, \dots, a_n$ is a list of pairwise distinct atoms, and $b_1, \dots, b_n$ too, then*

1. *Π_L is a permutation satisfying $\Pi_L(b_i) = a_i$, for $i = 1, \dots, n$,*
2. *$(a_1 \leftarrow b_1)\cdots(a_n \leftarrow b_n)$ and $(a_{\pi(1)} \leftarrow b_{\pi(1)})\cdots(a_{\pi(n)} \leftarrow b_{\pi(n)})$ are equivalent, for any permutation π.*
3. *For any $a, b \in \mathbb{A}$, $a =_L b$ iff $\Pi_L(a) = b$.*

Proof. By induction on n. For any $i = 1, \dots, n$, we have

$$\Pi_{(a_1\leftarrow b_1)\cdots(a_n\leftarrow b_n)} b_i = (a_1\ \Pi_{(a_2\leftarrow b_2)\cdots(a_n\leftarrow b_n)} b_1) \cdots \underbrace{(a_i\ \Pi_{(a_{i+1}\leftarrow b_{i+1})\cdots(a_n\leftarrow b_n)} b_i) \overbrace{\quad\cdots\quad (a_n b_n)}^{=\Pi_{(a_{i+1}\leftarrow b_{i+1})\cdots(a_n\leftarrow b_n)}} b_i}_{=a_i}$$

Hence, the i-th swapping changes $\Pi_{(a_{i+1}\leftarrow b_{i+1})\cdots(a_n\leftarrow b_n)} b_i$ by a_i. Now we are going to prove that a_i is not affected by the swappings $(a_j\ \Pi_{(a_{j+1}\leftarrow b_{j+1})\cdots(a_n\leftarrow b_n)} b_j)$ where $j > i$. On one hand, by assumption, $a_j \neq a_i$ when $j > i$. On the other hand, $\Pi_{(a_{j+1}\leftarrow b_{j+1})\cdots(a_n\leftarrow b_n)} b_j \neq a_i$ because $(a_{j+1} \leftarrow b_{j+1})\cdots(a_n \leftarrow b_n)$ is a strictly shorter replacing, and $i \in \{j+1, \dots, n\}$, therefore by induction hypothesis $(\Pi_{(a_{j+1}\leftarrow b_{j+1})\cdots(a_n\leftarrow b_n)})^{-1}(a_i) = b_i \neq b_j$. ∎

Lemma 5.9. *The replacing $L(a \leftarrow b)L'$ where a occurs on the left in L, and b occurs on the right in L, is equivalent to $L\,L'$. In other words, $L_1(a \leftarrow c)L_2(d \leftarrow b)L_3(a \leftarrow b)L_4$ and $L_1(a \leftarrow c)L_2(d \leftarrow b)L_3\,L_4$ are equivalent.*

If a occurs on the left in L, but b does not occur in the right in L, then, for any pair of terms t and u, $t =_{L(a\leftarrow b)L'} u$ iff $b\#u$ and $t =_{LL'} u$.

Similarly, if a does not occur on the left in L, but b occurs in the right in L, then, for any pair of terms t and u, $t =_{L(a\leftarrow b)L'} u$ iff $a\#t$ and $t =_{LL'} u$.

Proof. In nominal logic, and in λ-calculus we have the following implications:

$$\text{If } a\#t \text{ and } a.t \approx b.u, \text{ then } b\#u \text{ and } t \approx u \tag{5.1}$$

$$\text{If } t \approx u,\ a\#t \text{ and } b\#u, \text{ then } a.t \approx b.u \tag{5.2}$$

By definition of replacing, $t =_{(a_1\leftarrow b_1)\dots(a_n\leftarrow b_n)} u$ is equivalent to $a_n. \cdots .a_1.t \approx b_n. \cdots .b_1.u$.

For the first statement: For a given i, if $a_i \in \{a_{i-1}, \dots, a_1\}$, then $a_i\#a_{i-1}. \cdots .a_1.t$ and $t =_{(a_1\leftarrow b_1)\dots(a_n\leftarrow b_n)} u$ (using 5.1) imply $a_{i-1}. \cdots .a_1.t \approx b_{i-1}. \cdots .b_1.u$, hence $t =_{(a_1\leftarrow b_1)\dots(a_{i-1}\leftarrow b_{i-1})(a_{i+1}\leftarrow b_{i+1})\dots(a_n\leftarrow b_n)} u$. ∎

[2]Notice that we still have to ensure the freshness conditions

Lemmas 5.8 and 5.9 describe a characterization of replacings in terms of a set of pairs of atoms (normalized replacing), and a set of freshness equations. In the following we make explicit this characterization in terms of a set of pairs, called *rewriting set*, and a set of *forbidden atoms*.

Definition 5.10. Given a replacing L, we define the sets of rewriting pairs and forbidden atoms, noted $Rew(L)$ and $For(L)$, as follows

$$Rew(L) = \{(a \leftarrow b) \in \mathbb{A} \times \mathbb{A} \mid a \neq b \wedge a =_L b\}$$
$$For(L) = \{a \in \mathbb{A} \mid \neg(a =_L a)\}$$

Lemma 5.11. *Replacings L and L' are equivalent iff $Rew(L) = Rew(L')$ and $For(L) = For(L')$.*

Lemma 5.12. *For any replacing L we have*

$$Rew([\,]) = \emptyset$$
$$Rew(L(a \leftarrow b)) = \begin{cases} Rew(L) \cup \{a \leftarrow b\} & \text{if } a \neq b \text{ and } \forall c.a \leftarrow c \notin Rew(L) \text{ and } \forall c.c \leftarrow b \notin Rew(L) \\ Rew(L) & \text{otherwise} \end{cases}$$
$$For([\,]) = \emptyset$$
$$For(L(a \leftarrow b)) = \begin{cases} For(L) \cup \{b\} & \text{if } \exists c.a \leftarrow c \in Rew(L) \text{ and } \forall d.d \leftarrow b \notin Rew(L) \\ For(L) \cup \{a\} & \text{if } \exists d.d \leftarrow b \in Rew(L) \text{ and } \forall c.a \leftarrow c \notin Rew(L) \\ For(L) & \text{otherwise} \end{cases}$$

Proof. Given a replacing, we can use Lemma 5.9 to remove pairs with a duplicated component wisely until we obtain a normalized replacing. By Lemma 5.8, this normalized replacing is the rewriting set, whereas the set of freshness equations define the set of forbidden atoms. Then we can check that the previous recursions hold. ■

6. Generalized Replacings

Sometimes, simple replacings are not enough to represent the equations between atoms that we have to check. In some cases, we have to use a kind of *composition* of replacings. In this section we show how the notion of simple replacing may be generalized for this purpose, and how we can extend the definition of set of rewritings and set of forbidden atoms.

Definition 6.1. A generalized replacing is an expression generated by the grammar

$$L ::= Id \mid (a \leftarrow b) :: L \mid L_1 \circ L_2 \mid L^{-1}$$

with the following semantics

$t =_{Id} u$, if $t \approx u$,
$t =_{(a \leftarrow b)::L} u$, if $a.t =_L b.u$,
$t =_{L_1 \circ L_2} u$, if there exists a term v such that $t =_{L_1} v$ and $v =_{L_2} u$, and
$t =_{L^{-1}} u$, if $u =_L t$.

The sets $Rew(L)$ and $For(L)$ are defined for generalized replacings as for simple replacings.

Lemma 6.2. *Any generalized replacing is equivalent to a composition of simple replacings accordingly to the following equivalences between replacings*

$(L_1 \circ L_2) \circ L_3 = L_1 \circ (L_2 \circ L_3)$
$(a \leftarrow b) :: \big(L_1 \circ L_2\big) = \big((a \leftarrow b) :: L_1\big) \circ \big((a \leftarrow b) :: L_2\big)$
$(a_1 \leftarrow b_1) :: \cdots :: (a_n \leftarrow b_n) :: Id = (a_1 \leftarrow b_1) \cdots (a_n \leftarrow b_n)$

The following lemma shows us how we can recursively compute the set of rewritings and of forbidden atoms of a generalized replacing.

Lemma 6.3.

$$Rew(Id) = For(Id) = \emptyset$$

$$Rew\big((a \leftarrow b) :: L\big) = Rew(L) \setminus \{a \leftarrow c \mid \forall c \in \mathbb{A}\} \setminus \{c \leftarrow b \mid \forall c \in \mathbb{A}\} \cup \begin{cases} \{a \leftarrow b\} & \textit{if } a \neq b \\ \emptyset & \textit{if } a = b \end{cases}$$

$$For\big((a \leftarrow b) :: L\big) = For(L) \cup \{c \mid a \leftarrow c \in Rew(L) \vee c \leftarrow b \in Rew(L)\}$$

$$\begin{aligned} Rew(L_1 \circ L_2) = \; & \{a \leftarrow c \mid \exists b \in \mathbb{A}\; a \leftarrow b \in Rew(L_1) \wedge b \leftarrow c \in Rew(L_2)\} \\ & \cup \{a \leftarrow b \mid a \leftarrow b \in Rew(L_1) \wedge b \notin For(L_2)\} \\ & \cup \{a \leftarrow b \mid a \leftarrow b \in Rew(L_2) \wedge a \notin For(L_1)\} \end{aligned}$$

$$For(L_1 \circ L_2) = For(L_1) \cup For(L_2)$$

$$Rew(L^{-1}) = \{(b \leftarrow a) \mid (a \leftarrow b) \in Rew(L)\}$$

$$For(L^{-1}) = For(L)$$

7. A Paterson-Wegman Style Algorithm

In this section we describe our nominal unification algorithm in the style of Paterson and Wegman [Pat78], or, to be precise, in the style of the description that Martelli and Montanari [Mar82] makes of this algorithm.

First, w.l.o.g. we consider that we have a single nominal equation (we get rid of freshness equations, by Lemma 3.1, and reduce $\{t_1 \stackrel{?}{\approx} u_1, \dots, t_n \stackrel{?}{\approx} u_n\}$ to $f(t_1, \dots f(t_{n-1}, t_n) \dots) \stackrel{?}{\approx} f(u_1, \dots f(u_{n-1}, u_n) \dots)$, provided that there exists a binary constant f). Then, we flatten this equation, obtaining a set of equations of the form $X \stackrel{?}{\approx} f(Y_1, \dots, Y_n)$, $X \stackrel{?}{\approx} a.Y$, $X \stackrel{?}{\approx} a$ or $X \stackrel{?}{\approx} (a\,b)Y$, and a single equation $X_1 \stackrel{?}{\approx} X_2$, where X_1 and X_2 do not occur elsewhere bellow any other symbol. Finally, by Lemma 3.2, we can get rid of equations of the form $X \stackrel{?}{\approx} (a\,b)Y$. By Theorem 3.4, the resulting nominal unification problem has size $\mathcal{O}(|P|)$ on the size of the original problem.

Following the notation of [Mar82], equations of the form $X \stackrel{?}{\approx} f(Y_1, \dots, Y_n)$, $X \stackrel{?}{\approx} a.Y$, and $X \stackrel{?}{\approx} a$ are written in the form $\{X\} = f(Y_1, \dots, Y_n)$, $\{X\} = a.Y$, and $\{X\} = a$, respectively. The equation $X_1 \stackrel{?}{\approx} X_2$ is written as $X_1 =_{Id} X_2$, using the replacing Id. Then, we apply the following transformation rules wisely, where the equation $X_1 =_L X_2$ is in all cases a maximal equation, in the sense of Definition 2.2. Like in the classical Paterson-Wegman algorithm, there always exists an equation satisfying this condition, and we can find this equation intelligently, such that the total time consumed by this search is linearly bounded on the size of the original problem (see [Pat78] for more details).

Definition 7.1. Consider the following set of transformation rules:
Application:

$$\left.\begin{array}{l} \{\Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots\} = f(Y_1, \dots, Y_m) \\ \{\Pi_{L_2} X_2, \Pi_{L'_2} X'_2, \dots\} = f(Z_1, \dots, Z_m) \\ X_1 =_L X_2 \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \left\{\begin{array}{l} \Pi_{L_1} X_1, \Pi_{L'_1} X'_1, \dots, \\ \Pi_{L_1 \circ L} X_2, \Pi_{L_1 \circ L \circ L_2^{-1} \circ L'_2} X'_2, \dots \end{array}\right\} = f(Y_1, \dots, Y_m) \\ Y_1 =_{L_1 \circ L \circ L_2^{-1}} Z_1 \\ \dots \\ Y_m =_{L_1 \circ L \circ L_2^{-1}} Z_m \end{array}\right.$$

Abstraction:

$$\left.\begin{array}{l}\{\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots\}=a.Y\\ \{\Pi_{L_2}X_2,\Pi_{L'_2}X'_2,\dots\}=b.Z\\ X_1=_L X_2\end{array}\right\}\Longrightarrow\left\{\begin{array}{l}\left\{\begin{array}{l}\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots,\\ \Pi_{L_1\circ L}X_2,\Pi_{L_1\circ L\circ L_2^{-1}\circ L'_2}X'_2,\dots\end{array}\right\}=a.Y\\ Y=_{(a\leftarrow b)::(L_1\circ L\circ L_2^{-1})}Z\end{array}\right.$$

Atom:

$$\left.\begin{array}{l}\{\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots\}=a\\ \{\Pi_{L_2}X_2,\Pi_{L'_2}X'_2,\dots\}=b\\ X_1=_L X_2\end{array}\right\}\Longrightarrow\left\{\begin{array}{l}\left\{\begin{array}{l}\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots,\\ \Pi_{L_1\circ L}X_2,\Pi_{L_1\circ L\circ L_2^{-1}\circ L'_2}X'_2,\dots\end{array}\right\}=a\\ a=_{L_1\circ L\circ L_2^{-1}}b\end{array}\right.$$

Variable:

$$\left.\begin{array}{l}\{\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots\}=t\\ \{\Pi_{L_2}X_2,\Pi_{L'_2}X'_2,\dots\}=\emptyset\\ X_1=_L X_2\end{array}\right\}\Longrightarrow\left\{\begin{array}{l}\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots\\ \Pi_{L_1\circ L}X_2,\Pi_{L_1\circ L\circ L_2^{-1}\circ L'_2}X'_2,\dots\end{array}\right\}=t$$

$$\left.\begin{array}{l}\{\Pi_{L_1}X_1,\Pi_{L'_1}X'_1,\dots\}=\emptyset\\ \{\Pi_{L_2}X_2,\Pi_{L'_2}X'_2,\dots\}=t\\ X_1=_L X_2\end{array}\right\}\Longrightarrow\left\{\begin{array}{l}\Pi_{L_2\circ L^{-1}}X_1,\Pi_{L_2\circ L^{-1}\circ L_1^{-1}\circ L'_1}X'_1,\dots\\ \Pi_{L_2}X_2,\Pi_{L'_2}X'_2,\dots\end{array}\right\}=t$$

Theorem 7.2. *Given a simplified nominal unification problem P, P is solvable if, and only if, the rules of Definition 7.1 transform the problem into a set of equations of the form*

$$\left\{\begin{array}{c}\{\Pi_{L_1^1}X_1^1,\dots,\Pi_{L_1^{r_1}}X_1^{r_1}\}=t_1\\ \dots\\ \{\Pi_{L_m^1}X_m^1,\dots,\Pi_{L_m^{r_m}}X_m^{r_m}\}=t_m\\ a_1=_{L_1}b_1\\ \dots\\ a_n=_{L_n}b_n\end{array}\right\}$$

where $\{a_1=_{L_1}b_1,\dots,a_n=_{L_n}b_n\}$ holds, and the equations $t_i=_{L_i^j}X_i^j$, for $i=1,\dots,m$ and $j=1,\dots,r_i$, are solvable.

When P is solvable, then set of equations $t_i=_{L_i^j}X_i^j$ encode a solution.

Moreover, the size of the DAG representing the new set of equations is $\mathcal{O}(|P|)$, and it can be obtained in time $\mathcal{O}(|P|)$.

Proof. Soundness and completeness results from the rules $\approx$-abst-1, $\approx$-abst-2, and $\approx$-fun and $\approx$-atom of [Urb04], conveniently written in terms of replacings. The transformations resemble Paterson-Wegman transformations (Definition 2.2), and the termination proof is based on the same ideas. Notice that some transformations duplicate some L's. Therefore, the linear bound only applies representing equations as DAGs. ∎

Example 7.3. The equation $a.b.X\overset{?}{\approx}b.b.X$ can be simplified as:

$$\big\{\{X\}=\emptyset,\quad\{Y_1\}=a.Y_3,\quad\{Y_2\}=b.Y_4,\quad\{Y_3\}=b.X,\quad\{Y_4\}=b.X,\quad Y_1=_{Id}Y_2\big\}$$

Applying twice the abstraction rule we obtain:

$$\big\{\{X\}=\emptyset,\quad\{Y_1,Y_2\}=a.Y_3,\quad\{Y_3,\Pi_{(a\leftarrow b)::Id}Y_4\}=b.X,\quad X=_{(b\leftarrow b)::(a\leftarrow b)::Id}X\big\}$$

One application of the variable rule gives us the simplified equations

$$\big\{\{Y_1,Y_2\}=a.Y_3,\quad\{Y_3,\Pi_{(a\leftarrow b)::Id}Y_4\}=b.X,\quad\{X,\Pi_{(b\leftarrow b)::(a\leftarrow b)::Id}X\}=\emptyset\big\}$$

Example 7.4. From $a.\underbrace{b.\underbrace{\underbrace{a}_{Y_4}}_{Y_3}}_{Y_1} \stackrel{?}{\approx} \underbrace{b.\underbrace{b.X}_{Y_5}}_{Y_2}$, we obtain

$$\begin{array}{l} \{Y_1, Y_2\} = a.Y_3 \\ \{Y_3, \Pi_{(a\leftarrow b)::Id} Y_5\} = b.Y_4 \\ \{Y_4, \Pi_{(b\leftarrow b)::(a\leftarrow b)::Id} X\} = a \end{array}.$$

8. Efficient Checking of Replacings

Using the algorithm described in Definition 7.1, we obtain a set of replacing equations of the form $a =_L b$, a set of equations of the form $\{\Pi_{L^1} X^1, \ldots, \Pi_{L^r} X^r\} = t$ that codify the solution, and a DAG that represents the generalized replacings L's. Now, we will describe how we can check the solvability of these equations in quadratic time.

The main idea is to compute, for every node of the DAG, the two sets $Rew(L)$ and $For(L)$, where L is the replacing represented by this node. We will use the values of these sets already computed for the descendants of the node. Therefore, we proceed from the leaves of the DAG to the roots. We assume that we have a total ordering on the atoms $\mathbb{A}$. For efficiency, we compute three lists for every node L: a list RL that contains the elements of $Rew(L)$ ordered by the first component, RR with the elements of $Rew(L)$ ordered by the second component, and an ordered list F with the elements of $For(L)$. Moreover, the lists RL and RR are doubly linked, such that knowing the position of an element $(a \leftarrow b)$ in RL, we can know its position in RR and vice versa. Lemma 6.3 describes how to compute these list. Just as an example, Figure 1 shows how to compute RL, RR and F for $L = L_1 \circ L_2$, being RL_i, RR_i and F_i, for $i = 1, 2$, the respective lists for L_i.

To check if a set of equations P of the form $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = t$ has solution, and what is this solution, we compute the set of atoms that cannot occur free in the instance of X, written $For(X)$. This computation aborts (using rule 5) if P is unsolvable.

Definition 8.1. Given a set of equations P, for every variable X, we compute $For(X)$ as the minimal set of atoms that satisfy all the following rules, or we abort.

(1) If P contains $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = t$,
then $\Pi_{L_j^{-1}}\big(\Pi_{L_i}\big(For(X_i)\big) \cup For(L_i)\big) \subseteq For(X_j)$, for $i \neq j = 1, \ldots, r$.

(2) If P contains $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = f(Y_1, \ldots, Y_m)$,
then $\Pi_{L_i}(For(X_i)) \cup For(L_i) \subseteq For(Y_j)$, for $i = 1, \ldots, r$, and $j = 1, \ldots, m$.

(3) If P contains $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = a.Y$,
then $\Pi_{L_i}(For(X_i)) \cup For(L_i) \setminus \{a\} \subseteq For(Y)$, for $i = 1, \ldots, r$.

(4) If P contains $\{\Pi_L X, \Pi_{L'} X, \ldots\} = t$ and $\Pi_L(a) \neq \Pi_{L'}(a)$, for some $a \in \mathbb{A}$, then $a \in For(X)$.

(5) If P contains $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = a$ and $a \in \Pi_{L_i}(For(X_i)) \cup For(L_i)$, for some $i = 1, \ldots, r$, then P is unsolvable and abort.

Lemma 8.2. *Given a set of equations of the form $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = t$, we can compute $For(X)$, for every variable X, or abort, in quadratic time on the size of the DAG-representation of the equations.*

Moreover, the solution encoded by the equations is $\{a\#X \mid a \in For(X)\}$.

Proof. At every node of the DAG representing a generalized replacing L, we compute $Rew(L)$ and $For(L)$, using the values $Rew(L_i)$ and $For(L_i)$ previously computed for

```
Input: RL1, RR1, F1, RL2, RR2, F2
Output: RL, RR, F

i1 := 1 ; i2 := 1 ; j1 := 1 ; j2 := 1
while i1 ≤ RR1.size() and i2 ≤ RL2.size() do
    let (a ← b) = RR1[i1] and (b' ← c) = RL1[i2]
    if b = b' then
        following the double links, change (a ← b) in RL1 by (a ← c)
        following the double links, change (b ← c) in RR2 by (a ← c)
        remove (a ← b) from RR1 and (b ← c) from RL2
        i1 := i1 + 1
        i2 := i2 + 1
    else if b < b' then
        while j2 ≤ F2.size() and F2[j2] < b do  j2 := j2 + 1
        if j2 ≤ F2.size() and F2[j2] = b then
            remove (a ← b) from RR1 and RL1
        i1 := i1 + 1
    else while j1 ≤ F1.size() and F1[j1] < b do j1 := j1 + 1
        if j1 ≤ F1.size() and F1[j1] = b' then
            remove (b' ← c) from RR2 and RL2
        i2 := i2 + 1
if i1 = RR1.size() then
    while i2 ≤ RL2.size() do
        while j1 ≤ F1.size() and F1[j1] < b' do j1 := j1 + 1
        if j1 ≤ F1.size() and F1[j1] = b' then
            remove (b' ← c) from RR2 and RL2
        i2 := i2 + 1
else while i1 ≤ RR1.size() do
    while j2 ≤ F2.size() and F2[j2] < b do j2 := j2 + 1
    if j2 ≤ F2.size() and F2[j2] = b then
        remove (a ← b) from RR1 and RL1
    i1 := i1 + 1
RL := merge(RL1, RL2)
RR := merge(RR1, RR2)
F := merge(F1, F2)
return RR, RL, F
```

Figure 1: Computation of $Rew(L_1 \circ L_2)$ and $For(L_1 \circ L_2)$ in time $\mathcal{O}(|Rew(L_1)|+|Rew(L_2)|+|For(L_1)|+|For(L_2)|)$.

the descendants L_i of the node. This computation takes at worst linear time for every node, being the worst case the composition of replacings $L = L_1 \circ L_2$ with time $\mathcal{O}(|Rew(L_1)| + |Rew(L_2)| + |For(L_1)| + |For(L_2)|)$, described in Figure 1. Therefore, the overall computation takes quadratic time. Then, using the rules of Definition 8.1, in quadratic time we can check if all equations are solvable. ∎

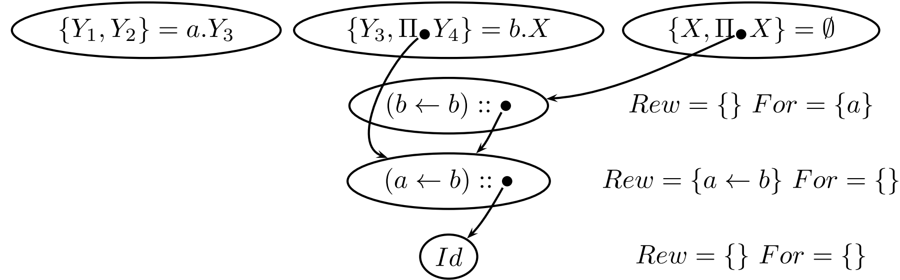
Theorem 8.3. *Nominal Unification can be decided in quadratic time.*

Proof. By Theorem 3.4 we can assume that nominal equations are simplified. Then, by Theorem 7.2, we can transform these equations into an equivalent set of equations of the form $a =_L b$ or $\{\Pi_{L_1} X_1, \dots, \Pi_{L_r} X_r\} = t$ represented as a DAG, in linear time on the

size of the original equations. Equations $a =_L b$ are solvable if $(a \leftarrow b) \in Rew(L)$. By Lemma 8.2, we can compute $For(X)$ for every variable, checking the solvability of equations $\{\Pi_{L_1} X_1, \ldots, \Pi_{L_r} X_r\} = t$. ■

Example 8.4. Consider example 7.3, where we obtain
$\big\{\{Y_1, Y_2\} = a.Y_3, \quad \{Y_3, \Pi_{(a\leftarrow b)::Id} Y_4\} = b.X, \quad \{X, \Pi_{(b\leftarrow b)::(a\leftarrow b)::Id} X\} = \emptyset\big\}$.

The DAG representation with $Rew(L)$ and $For(L)$ of every node representing a generalized replacing is as follows.



Definition 8.1 computes $For(Y_3) = For(X) = \{a\}$, $For(Y_4) = \{b\}$, $For(Y_1) = For(Y_2) = \emptyset$. Now, considering only original variables, i.e. X, we obtain the solution $a\#X$.

In example 7.4, the equation $\{Y_4, \Pi_{(b\leftarrow b)::(a\leftarrow b)::Id} X\} = a$, using rule 5 of Definition 8.1, allows us to deduce that the problem is unsolvable.

9. Conclusions, can we do it better?

We have presented an efficient algorithm that computes nominal unifiers in quadratic time. This result does not improve the bound found by ourself by reduction to the problem of Higher-Order Pattern Unification [Lev08]. The natural question now is: can we still improve this bound?

A careful analysis of the algorithm of Figure 1 shows us that it is basically a merge function, and that the complete check of the whole DAG of replacings is not very distinct from a merge-sort algorithm. In fact, if we could ensure that, when $L = L_1 \circ L_2$, we have $|Rew(L)| + |For(L)| \geq |Rew(L_1)| + |For(L_1)| + |Rew(L_2)| + |For(L_2)|$ and $|Rew(L_1)| + |For(L_1)| \approx |Rew(L_2)| + |For(L_2)|$, then the cost of the algorithm would be dominated by $T(n) = 2\,T(n/2) + \mathcal{O}(n)$ that has solution $\mathcal{O}(n \log n)$. If we could ensure $|Rew(L)| + |For(L)| \geq |Rew(L_1)| + |For(L_1)| + |Rew(L_2)| + |For(L_2)|$, but not the balance between the data structures of L_1 and L_2, then we could implement the sorted lists using AVL, and apply the ideas of Brown and Tarjan [Bro79] for merging of unbalanced sorted lists. This unbalance merge of two lists of sizes n_1 and n_2 can be done in time $\mathcal{O}(n_1 \log \frac{n_2}{n_1})$. Therefore, the time of the complete checking would be dominated by $T(n) = T(n_1) + T(n_2) + \mathcal{O}(n_1 \log \frac{n_2}{n_1})$, where $n = n_1 + n_2$. In this case, the solution is also $\mathcal{O}(n \log n)$. Therefore, we can conclude that we can check a set of replacings in time $\mathcal{O}(n \log n)$ on the size of the *tree* (not the DAG) representing the replacing. This means that, when the DAG is a tree, for instance in example 4.2, we can check the replacings in quasi-linear time.

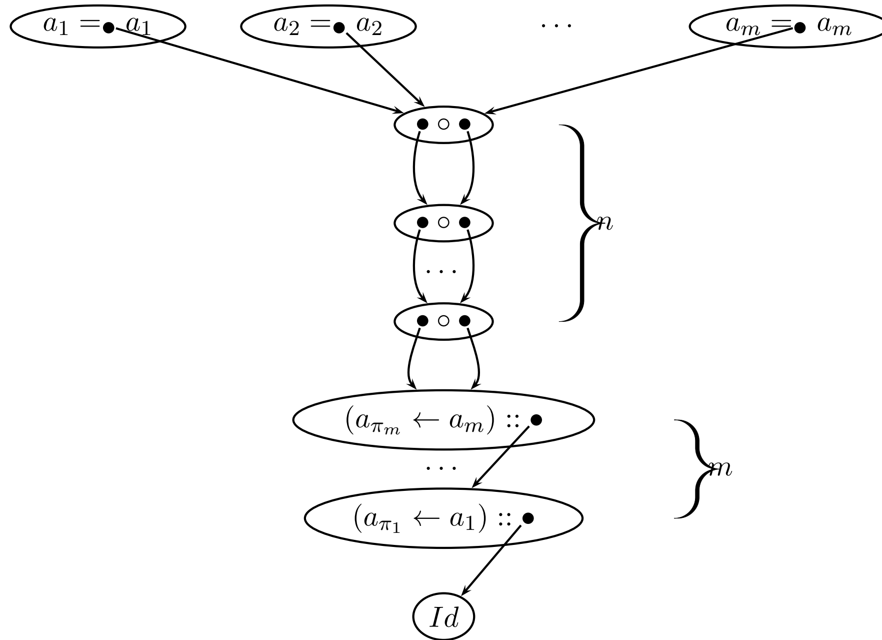
To conclude, consider the following example, that shows that the quadratic bound seems difficult to improve in the general case.

Example 9.1. Given a permutation π of $m = |\pi|$ elements, an a value n, we can construct the following two equation of size $\mathcal{O}(n+m)$

$$
\begin{gathered}
a_{\pi_1}.\cdots.a_{\pi_m}.f(f(\dots f(Y,X_n)\dots,X_2),X_1) \approx \\
a_1.\cdots.a_m.f(X_1,f(X_2,\dots f(X_n,Y)\dots))
\end{gathered}
$$

$$
Y \approx f(a_1, f(a_2, \dots f(a_{m-1}, a_m)\dots))
$$

From these equations we get the following DAG. This problem is solvable, if we have $\pi^{2^n} = Id$. It seems difficult to answer this question in time faster than $\mathcal{O}(n\,m)$.



Appendix

Example 9.2 (Cont. of Example 4.2)**.** In fact, we would obtain the same result from the lazy application of the transformation rules of the nominal unification algorithm from [Urb04]. Those rules basically encode the inference rules presented in Section 2, namely we use here the ones for $\approx$-*abst-1* and $\approx$-*abst-2*. What we do is to delay the check for equality or difference between atoms of two abstractions because one of them can have a permutation applied on it, and we don't want to compute permutations until the end. By default we apply the rule for $\approx$-*abst-2* constraining the freshness predicate to the proviso of difference between abstractions.

$$\begin{array}{l}
a_2.a_1.f(c_1,c_2) \overset{?}{\approx} (a_3 b_3) b_2.b_1.f(d_1,d_2)\\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2)\\
\hline
\\
a_1.f(c_1,c_2) \overset{?}{\approx} (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1.f(d_1,d_2)\\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2)\\
a_2 \neq (a_3 b_3) b_2 \implies a_2 \# (a_3 b_3) b_1.f(d_1,d_2)\\
\hline
\\
f(c_1,c_2) \overset{?}{\approx} (a_1 (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1)(a_2 (a_3 b_3) b_2)(a_3 b_3) f(d_1,d_2)\\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2)\\
a_2 \neq (a_3 b_3) b_2 \implies a_2 \# (a_3 b_3) b_1.f(d_1,d_2)\\
a_1 \neq (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1 \implies a_1 \# (a_2 (a_3 b_3) b_2)(a_3 b_3) f(d_1,d_2)\\
\hline
\\
c_1 \overset{?}{\approx} (a_1 (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1)(a_2 (a_3 b_3) b_2)(a_3 b_3) d_1\\
c_2 \overset{?}{\approx} (a_1 (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1)(a_2 (a_3 b_3) b_2)(a_3 b_3) d_2\\
a_3 \neq b_3 \implies a_3 \# b_2.b_1.f(d_1,d_2)\\
a_2 \neq (a_3 b_3) b_2 \implies a_2 \# (a_3 b_3) b_1.f(d_1,d_2)\\
a_1 \neq (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1 \implies a_1 \# (a_2 (a_3 b_3) b_2)(a_3 b_3) f(d_1,d_2)
\end{array}$$

Now, we get rid of the freshness constraints, translating them into disequalities by means of rules of the nominal unification algorithm from [Urb04] for the freshness predicates of Section 2.

$$\begin{array}{l}
c_1 \overset{?}{\approx} (a_1 (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1)(a_2 (a_3 b_3) b_2)(a_3 b_3) d_1\\
c_2 \overset{?}{\approx} (a_1 (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1)(a_2 (a_3 b_3) b_2)(a_3 b_3) d_2\\
a_3 \neq b_3 \wedge a_3 \neq b_2 \wedge a_3 \neq b_1 \implies (a_3 \neq d_1 \wedge a_3 \neq d_2)\\
a_2 \neq (a_3 b_3) b_2 \wedge a_2 \neq (a_3 b_3) b_1 \implies (a_2 \neq (a_3 b_3) d_1 \wedge a_2 \neq (a_3 b_3) d_2)\\
a_1 \neq (a_2 (a_3 b_3) b_2)(a_3 b_3) b_1 \implies (a_1 \neq (a_2 (a_3 b_3) b_2)(a_3 b_3) d_1 \wedge a_1 \neq (a_2 (a_3 b_3) b_2)(a_3 b_3) d_2)
\end{array}$$

References

[Bro79] Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. of the ACM*, 26(2):211–226, 1979.

[Cal07] Christophe Calvès and Maribel Fernández. Implementing nominal unification. *ENTCS*, 176(1):25–37, 2007.

[Cal08] Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theoretical Computer Science*, 403(2-3):285–306, 2008.

[Cal10] Christophe Calvès. *Complexity and Implementation of Nominal Algorithms.* Ph.D. thesis, King's College London, 2010.

[Che04] James Cheney and Christian Urban. α-prolog: A logic programming language with names, binding and α-equivalence. In *Proc. of the 20th Int. Conf. on Logic Programming, ICLP'04*, *LNCS*, vol. 3132, pp. 269–283. 2004.

[Che05] James Cheney. Equivariant unification. In *Proc. of the 16th Int. Conf. on Term Rewriting and Applications, RTA'05*, *LNCS*, vol. 3467, pp. 74–89. 2005.

[Clo07] R. Clouston and A. Pitts. Nominal equational logic. *ENTCS*, 1496:223–257, 2007.

[Dow09] Gilles Dowek, Murdoch Gabbay, and Dominic Mulligan. Permissive nominal terms and their unification. In *Proc. of the 24th Convegno Italiano di Logica Computazionale, CILC'09*. 2009.

[Dow10] Gilles Dowek, Murdoch Gabbay, and Dominic Mulligan. Permissive nominal terms and their unification. *Logic Journal of the IGPL*, 2010.

[Fer05] Maribel Fernández and Murdoch Gabbay. Nominal rewriting with name generation: abstraction vs. locality. In *Proc. of the 7th Int. Conf. on Principles and Practice of Declarative Programming, PPDP'05*, pp. 47–58. 2005.

[Fer07] Maribel Fernández and Murdoch Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, 2007.

[Gab01] Murdoch Gabbay and A. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, 2001.

[Gab06] Murdoch Gabbay and Aad Mathijssen. Nominal algebra. In *Proc. of the 18th Nordic Workshop on Programming Theory, NWPT'06*. 2006.

[Gab07] Murdoch Gabbay and Aad Mathijssen. A formal calculus for informal equality with binding. In *Logic, Language, Information and Computation*, *LNCS*, vol. 4576, pp. 162–176. Springer, 2007.

[Gab09] Murdoch Gabbay and Aad Mathijssen. Nominal (universal) algebra: equational logic with names and binding. *Journal of Logic and Computation*, 19(6):1455–1508, 2009.

[Lev08] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In *Proc. of the 19th Int. Conf on Rewriting Techniques and Applications, RTA'08*, *LNCS*, vol. 5117, pp. 246–260. 2008.

[Mar82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[Pat78] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

[Pit01] Andrew Pitts. Nominal logic: A first order theory of names and binding. In *Proc. of the 4th Int. Symp. on Theoretical Aspects of Computer Software, TACS'01*, *LNCS*, vol. 2215, pp. 219–242. 2001.

[Pit03] A. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[Qia96] Zhenyu Qian. Unification of higher-order patterns in linear time and space. *J. of Logic and Computation*, 6(3):315–341, 1996.

[Urb03] C. Urban, A. Pitts, and M. Gabbay. Nominal unification. In *Proc. of the 17th Int. Work. on Computer Science Logic, CSL'03*, *LNCS*, vol. 2803, pp. 513–527. 2003.

[Urb04] C. Urban, A. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

[Urb05] Christian Urban and James Cheney. Avoiding equivariance in alpha-prolog. In *Proc. of the Int. Conf. on Typed Lambda Calculus and Applications, TLCA'05*, *LNCS*, vol. 3461, pp. 401–416. 2005.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 227-242
http://rewriting.loria.fr/rta/

COMPUTING CRITICAL PAIRS IN 2-DIMENSIONAL REWRITING SYSTEMS

SAMUEL MIMRAM

CEA, LIST, Point Courrier 94, 91191 Gif-sur-Yvette, France.
E-mail address: samuel.mimram@cea.fr

Abstract. Rewriting systems on words are very useful in the study of monoids. In good cases, they give finite presentations of the monoids, allowing their manipulation by a computer. Even better, when the presentation is confluent and terminating, they provide one with a notion of canonical representative for the elements of the presented monoid. Polygraphs are a higher-dimensional generalization of this notion of presentation, from the setting of monoids to the much more general setting of n-categories. Here, we are interested in proving confluence for polygraphs presenting 2-categories, which can be seen as a generalization of term rewriting systems. For this purpose, we propose an adaptation of the usual algorithm for computing critical pairs. Interestingly, this framework is much richer than term rewriting systems and requires the elaboration of a new theoretical framework for representing critical pairs, based on contexts in compact 2-categories.

Term rewriting systems have proven very useful to reason about terms modulo equations. In some cases, the equations can be oriented and completed in a way giving rise to a converging (i.e. confluent and terminating) rewriting system, thus providing a notion of canonical representative of equivalence classes of terms. Usually, terms are freely generated by a *signature* $(\Sigma_n)_{n\in\mathbb{N}}$, which consists of a family of sets Σ_n of generators of arity n, and one considers *equational theories* on such a signature, which are formalized by sets of pairs of terms called *equations*. For example, the equational theory of monoids contains two generators m and e, whose arities are respectively 2 and 0, and three equations

$$m(m(x,y),z) = m(x,m(y,z)) \qquad m(e,x) = x \quad \text{and} \quad m(x,e) = x$$

These equations, when oriented from left to right, form a rewriting system which is converging. The termination of this system can be shown by giving an interpretation of the terms in a well-founded poset, such that the rewriting rules are strictly decreasing. Since the system is terminating, the confluence can be deduced from the local confluence, which can itself be shown by verifying that the five critical pairs

$$m(m(m(x,y),z),t) \qquad m(m(e,x),y) \qquad m(m(x,e),y) \qquad m(m(x,y),e) \qquad m(e,e)$$

are joinable and these critical pairs can be computed using a unification algorithm. A more detailed presentation of term rewriting systems along with the classic techniques to prove their convergence can be found in [Baa99].

This work was started while I was in the PPS team (CNRS – Univ. Paris Diderot) and has been supported by the CHOCO (“Curry Howard pour la Concurrence”, ANR-07-BLAN-0324) French ANR project.

© S. Mimram
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.227

As a particular case, when the generators of an equational theory are of arity one, the category of terms modulo the congruence generated by the equations is a monoid, with addition given by composition and neutral element being the identity. A *presentation* of a monoid $(M, \times, 1)$ is such an equational theory, which is generating a monoid isomorphic to M. For example the monoid $\mathbb{N}/2\mathbb{N}$ is presented by the equational theory with only one generator a, of arity one, and the equation $a(a(x)) = x$. Presentations of monoids are particularly useful since they can provide finite description of monoids which may be infinite, thus allowing their manipulation with a computer. More generally, with generators of any arity, equational theories give rise to presentations of Lawvere theories [Law63], which are cartesian categories whose objects are the natural integers and such that product is given on objects by addition: a signature namely generates such a category, whose morphisms $f : m \to n$ are n-uples of terms with m free variables, composition being given by substitution.

Term rewriting systems have been generalized by *polygraphs*, in order to provide a formal framework in which one can give presentations of any (strict) n-category. We are interested here in adapting the classical technique to study confluence of 3-polygraphs, which give rise to presentations of 2-categories, by computing their critical pairs. These polygraphs can be seen as term rewriting systems improved on the following points:

- the variables of terms are simply typed (this can be thought as generalizing from a Lawvere theory of terms to any cartesian category of terms),
- variables in terms cannot necessarily be duplicated, erased or swapped (the categories of terms are not necessarily cartesian but only monoidal),
- and the terms can have multiple outputs as well as multiple inputs.

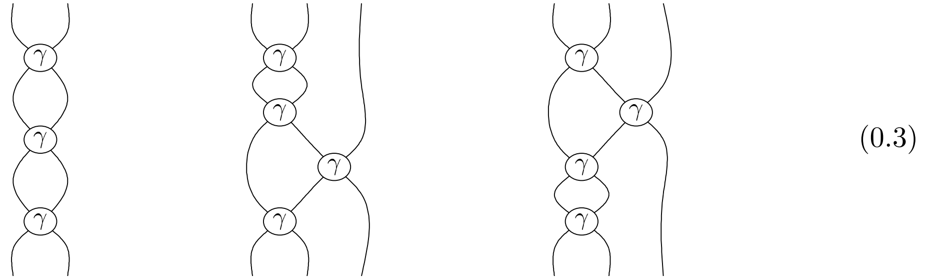
Many examples of presentations of monoidal categories where studied by Lafont [Laf03], Guiraud [Gui06b, Gui06a] and the author [Mim08, Mim09b]. A fundamental example is the 3-polygraph S, presenting the monoidal category **Bij** (the category of finite ordinals and bijections). This polygraph has one generator for objects 1, one generator for morphisms $\gamma : 2 \to 2$ (where 2 is a notation for $1 \otimes 1$) and two equations

$$(\gamma \otimes 1) \circ (1 \otimes \gamma) \circ (\gamma \otimes 1) = (1 \otimes \gamma) \circ (\gamma \otimes 1) \circ (1 \otimes \gamma) \qquad \text{and} \qquad \gamma \circ \gamma = 1 \otimes 1 \quad (0.1)$$

where the morphism 1 is a short notation for id_1. That this polygraph is a presentation of the category **Bij** means that this category is isomorphic to the free monoidal category containing an object 1 and a generator γ, quotiented by the smallest congruence generated by the equations (0.1). This result can be seen as a generalization of the presentation of the symmetric groups by transpositions. These equations can be better understood with the graphical notation provided by *string diagrams*, which is a diagrammatic notation for morphisms in monoidal categories, introduced formally in [Joy91]. The morphism γ should be thought as a device with two inputs and two outputs of type 1, and the two equations (0.1) can thus be represented graphically by

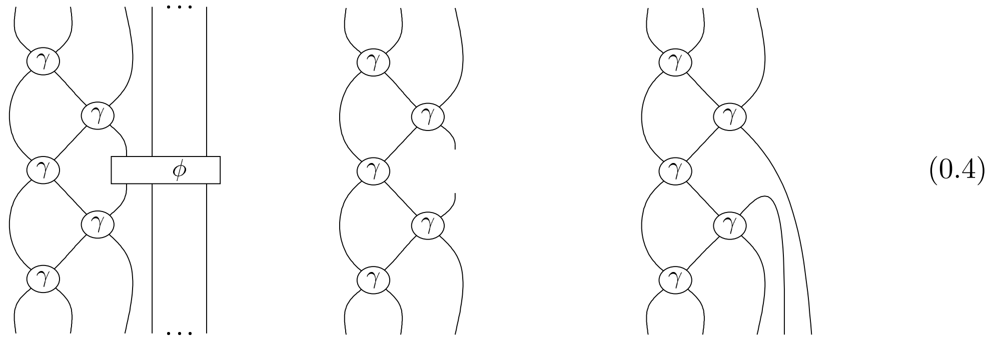
$\gamma\ \gamma\ \gamma = \gamma\ \gamma\ \gamma$ and $\gamma\ \gamma =$ (0.2)

In this notation, wires represent identities (on the object 1), horizontal juxtaposition of diagrams corresponds to tensoring, and vertical linking of diagrams corresponds to composition of morphisms. Moreover, these diagrams should be considered modulo planar continuous deformations, so that the axioms of monoidal categories are verified. These diagrams are conceptually important because they allow us to see morphisms in monoidal categories either as algebraic objects or as geometric objects (some sort of planar graphs). If we orient both equations from left to right, we get a rewriting system which can be shown to be convergent. It has the three following critical pairs [Laf03]:



(0.3)

Moreover, for every morphism $\phi : 1 \otimes m \to 1 \otimes n$, the morphism on the left of (0.4)



(0.4)

can be rewritten in two different ways, thus giving rise to an infinite number of critical pairs for the rewriting system. This phenomenon was first observed by Lafont [Laf03] and later on studied by Guiraud and Malbos [Gui09]. Interestingly, we can nevertheless consider that there is a finite number of critical pairs if we allow ourselves to consider the "diagram" on the center of (0.4) as a critical pair. Of course, this diagram does not make sense at first. However, we can give a precise meaning to it if we embed our terms in a larger category, which is compact: in such a category every object has a dual, which corresponds graphically to having the ability to bend wires (see the figure on the right). This observation was the starting point of this paper which is devoted to formalizing these intuitions in order to propose an algorithm for computing critical pairs in polygraphs.

We believe that this is a major area of higher-dimensional algebra where computer scientists should step in: typical presentations of categories can give rise to a very large number of critical pairs and having automated tools to compute them seems to be necessary in order to push further the study of those systems. The present paper constitutes a first step in this direction, by defining the structures necessary to manipulate algorithmically the morphisms in categories generated by polygraphs and by proposing an algorithm to compute the critical pairs in polygraphic rewriting systems. Conversely, algebra provides strong indications about technical choices that should be made in order to generalize rewriting theory in higher dimensions. We have done our possible to provide an overview of the

theoretical tools used here, as well as intuitions about them. A preliminary detailed version of this work is available in [Mim09a].

We begin by recalling the definition of polygraphs, describe the categories they generate, and formulate the unification problem in this framework using the notion of context in a 2-category. Then, we show that 2-categories can be fully and faithfully embedded into the free compact 2-category they generate, which allows us to describe a unification algorithm for polygraphic rewriting systems.

1. Presentations of 2-categories

Because of space limitations, we have to omit the basic definitions in category theory and refer the reader to MacLane's reference book [Mac71]. We only recall that a *2-category* is a generalization in dimension 2 of the concept of category. It consists essentially of a class of 0-*cells* A, a class of 1-*cells* $f : A \to B$ (with 0-cells A and B as source and target) and a class of 2-*cells* $\alpha : f \Rightarrow g : A \to B$ (with parallel 1-cells $f : A \to B$ and $g : A \to B$ as source and target), together with a *vertical composition*, which to every pair of 2-cells $\alpha : f \Rightarrow g$ and $\beta : g \Rightarrow h$ associates a 2-cell $\beta \circ \alpha : f \Rightarrow h$, and a *horizontal composition*, which to every pair of 2-cells $\alpha : f \Rightarrow g$ and $\beta : h \Rightarrow i$ associates a 2-cell $\alpha \otimes \beta : (f \otimes h) \Rightarrow (g \otimes i)$, such that vertical and horizontal composition are associative, admit neutral elements (the identities) and the *exchange law* is satisfied: for every four 2-cells

$$\alpha : f \Rightarrow f' : A \to B, \quad \alpha' : f' \Rightarrow f'' : A \to B, \quad \beta : g \Rightarrow g' : B \to C, \quad \beta' : g' \Rightarrow g'' : B \to C$$

the following equality holds

$$(\alpha' \circ \alpha) \otimes (\beta' \circ \beta) \quad = \quad (\alpha' \otimes \beta') \circ (\alpha \otimes \beta) \tag{1.1}$$

as well as a nullary version of this law: $\mathrm{id}_{A \otimes B} = \mathrm{id}_A \otimes \mathrm{id}_B$ for every objects A and B. In a 2-category, two n-cells are *parallel* when they have the same source and the same target. We also recall that two 0-cells A and B of a 2-category $\mathcal{C}$, induce a category $\mathcal{C}(A, B)$, called *hom-category*, whose objects are the 1-cells $f : A \to B$ of $\mathcal{C}$ and whose morphisms $\alpha : f \Rightarrow g$ are 2-cells of $\mathcal{C}$, composition being given by vertical composition. A (strict) *monoidal category* is a 2-category with exactly one 0-cell.

Polygraphs are algebraic structures which were introduced in their 2-dimensional version by Street [Str76] under the name *computads*, later on generalized to higher dimensions by Power [Pow90], and independently rediscovered by Burroni [Bur93]. We are specifically interested in 3-polygraphs, which give rise to presentations of 2-categories, and briefly recall their definition here. This definition is a bit technical but conceptually clear: it consists of sets of 0-, 1-, 2-generators for "terms", each 2-generator having a list of 1-generators as source and as target, each 1-generator having itself a 0-generator as source and as target, together with a set of equations which are pairs of terms (generated by the 2-generators).

Suppose that we are given a set E_0 of *0-generators*, such a set will be called a *0-polygraph.* We write $E_0^* = E_0$ and $i_0 : E_0 \to E_0^*$ the identity function. A 1-polygraph on these generators is a graph, that is a diagram $E_0^* \underset{t_0}{\overset{s_0}{\leftleftarrows}} E_1$ in **Set**, with E_0^* as vertices, the elements of E_1 being called *1-generators*. We can construct a free category on this graph: its set E_1^* of morphisms is the set of paths in the graph (identities are the empty paths), the source $s_0^*(f)$ (resp. target $t_0^*(f)$) of a morphism $f \in E_1^*$ being the source (resp. target) of the path. If we write $i_1 : E_1 \to E_1^*$ for the injection of the 1-generators into morphisms of

this category, which to every 1-generator associates the corresponding path of length one, we thus get a diagram

$$\begin{array}{ccc} E_0 & & E_1 \\ {\scriptstyle i_0}\downarrow & \swarrow\!\swarrow{\scriptstyle s_0,\, t_0} & \downarrow{\scriptstyle i_1} \\ E_0^* & \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} & E_1^* \end{array} \tag{1.2}$$

in **Set**, which is commutative in the sense that $s_0^* \circ i_1 = s_0$ and $t_0^* \circ i_1 = t_0$. A *2-polygraph* on this 1-polygraph consists of a diagram

$$\begin{array}{ccccc} E_0 & & E_1 & & E_2 \\ {\scriptstyle i_0}\downarrow & \swarrow\!\swarrow{\scriptstyle s_0,\, t_0} & \downarrow{\scriptstyle i_1} & \swarrow\!\swarrow{\scriptstyle s_1,\, t_1} & \\ E_0^* & \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} & E_1^* & & \end{array} \tag{1.3}$$

in **Set**, such that $s_0^* \circ s_1 = s_0^* \circ t_1$ and $t_0^* \circ s_1 = t_0^* \circ t_1$. The elements of E_2 are called *2-generators*. Again we can generate a free 2-category on this data, whose underlying category is the category generated in (1.2) and which has the 2-generators as morphisms. If we write E_2^* for its set of morphisms and $i_2 : E_2 \to E_2^*$ for the injection of the 2-generators into morphisms, we thus get a diagram

$$\begin{array}{ccccc} E_0 & & E_1 & & E_2 \\ {\scriptstyle i_0}\downarrow & \swarrow\!\swarrow{\scriptstyle s_0,\, t_0} & \downarrow{\scriptstyle i_1} & \swarrow\!\swarrow{\scriptstyle s_1,\, t_1} & \downarrow{\scriptstyle i_2} \\ E_0^* & \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} & E_1^* & \overset{s_1^*}{\underset{t_1^*}{\leftleftarrows}} & E_2^* \end{array} \tag{1.4}$$

We can now formulate the definition of 3-polygraphs as follows.

Definition 1.1. A *3-polygraph* consists of a diagram

$$\begin{array}{ccccccc} E_0 & & E_1 & & E_2 & & E_3 \\ {\scriptstyle i_0}\downarrow & \swarrow\!\swarrow{\scriptstyle s_0,\, t_0} & \downarrow{\scriptstyle i_1} & \swarrow\!\swarrow{\scriptstyle s_1,\, t_1} & \downarrow{\scriptstyle i_2} & \swarrow\!\swarrow{\scriptstyle s_2,\, t_2} & \\ E_0^* & \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} & E_1^* & \overset{s_1^*}{\underset{t_1^*}{\leftleftarrows}} & E_2^* & & \end{array} \tag{1.5}$$

(where E_i^*, s_i^* and t_i^* are freely generated as previously explained), such that

$$s_i^* \circ s_{i+1} = s_i^* \circ t_{i+1} \quad \text{and} \quad t_i^* \circ s_{i+1} = t_i^* \circ t_{i+1}$$

for $i = 0$ and $i = 1$, together with a structure of 2-category on the 2-graph

$$E_0^* \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} E_1^* \overset{s_1^*}{\underset{t_1^*}{\leftleftarrows}} E_2^*$$

Again, a 3-polygraph freely generates a 3-category $\mathcal{C}$ whose underlying 2-category is the underlying 2-category of the polygraph and whose 3-cells are generated by the 3-generators of the polygraph. A quotient 2-category $\tilde{\mathcal{C}}$ can be constructed from this 2-category: it is defined as the underlying 2-category of $\mathcal{C}$ quotiented by the congruence identifying two 2-cells whenever there exists a 3-cell between them in $\mathcal{C}$. A 3-polygraph P *presents* a 2-category $\mathcal{D}$ when $\mathcal{D}$ is isomorphic to the 2-category $\tilde{\mathcal{C}}$ induced by the polygraph P. In this sense, the underlying 2-polygraph of a 3-polygraph is a *signature* generating terms which are

to be considered modulo the *equations* described by the 3-generators; these equations $r \in E_3$ being oriented, they will be called *rewriting rules*, the source $s_2(r)$ (resp. the target $t_2(r)$) being the *left member* (resp. *right member*) of the rule. A polygraph is *finite* when all the sets E_i are; in the following, we only consider such polygraphs.

A *morphism of polygraphs* F between two 3-polygraphs P and Q consists of a 4-uple (F_0, F_1, F_2, F_3) of functions $F_i : E_i^P \to E_i^Q$, such that the obvious diagrams commute (for example, for every i, $s_i^Q \circ F_{i+1} = F_i^* \circ s_i^P$, where $F_i^* : E_i^{P*} \to E_i^{Q*}$ is the monoid morphism induced by F_i). We write n-**Pol** for the category of n-polygraphs (this construction can be carried on to any dimension $n \in \mathbb{N}$ but we will only consider cases with $n \leqslant 3$). These categories have many nice properties, amongst which being cocomplete. The free n-category generated by an n-polygraph P is denoted $\mathcal{C}_n(P)$. Given an integer $k \leqslant n$, we write $U_k : n\text{-}\mathbf{Pol} \to k\text{-}\mathbf{Pol}$ for the forgetful functor which simply forgets about the sets of generators of dimension higher than k. This functor admits a left adjoint $F_n : k\text{-}\mathbf{Pol} \to n\text{-}\mathbf{Pol}$ which adds empty sets of generators of dimension higher than k. We sometimes leave implicit the inclusion of k-**Pol** into n-**Pol** induced by F_n.

Example 1.2. The theory of *symmetries* mentioned in the introduction is the polygraph S whose generators are

$$E_0 = \{*\} \qquad E_1 = \{1 : * \to *\} \qquad E_2 = \{\gamma : 1 \otimes 1 \Rightarrow 1 \otimes 1\}$$
$$E_3 = \{y : (\gamma \otimes 1) \circ (1 \otimes \gamma) \circ (\gamma \otimes 1) \Rrightarrow (1 \otimes \gamma) \circ (\gamma \otimes 1) \circ (1 \otimes \gamma),\ s : \gamma \circ \gamma \Rrightarrow 1 \otimes 1\}$$

Example 1.3. The theory of *monoids* is the polygraph M defined by

$$E_0 = \{*\} \qquad E_1 = \{1 : * \to *\} \qquad E_2 = \{\mu : 1 \otimes 1 \Rightarrow 1,\ \eta : * \Rightarrow 1\}$$
$$E_3 = \{a : \mu \circ (\mu \otimes 1) \Rrightarrow \mu \circ (1 \otimes \mu),\ l : \mu \circ (\eta \otimes 1) \Rrightarrow 1,\ r : (1 \otimes \eta) \to 1\}$$

This polygraph presents the augmented simplicial category (the category of finite ordinals and non-decreasing functions).

2. Formal representation of free 2-categories

The definition of 3-polygraphs involves the construction of free categories and free 2-categories, which are abstractly defined in category theory by universal constructions. Here, we need a more concrete representation of these mathematical objects. As already mentioned, the free category (1.2) on a graph is easy to describe: its objects are the vertices of the graph and morphisms are paths of the graph with composition given by concatenation. However, describing the free 2-category on a 2-polygraph in an effective way (which can be implemented) is much less straightforward. Of course, following the definition given in Section 1, one could describe the 2-cells of this 2-category as formal vertical and horizontal compositions of 2-generators up to a congruence imposing associativity and absorption of units for both compositions and the exchange law (1.1). However, given an object A in a 2-category $\mathcal{C}$ and two 2-cells $\alpha, \beta : \mathrm{id}_A \Rightarrow \mathrm{id}_A : A \to A$ of this category, the equality $\alpha \otimes \beta = \beta \otimes \alpha$ can be deduced from the following sequence of equalities:

$$\alpha\otimes\beta = (\mathrm{id}_A\circ\alpha)\otimes(\beta\circ\mathrm{id}_A) = (\mathrm{id}_A\otimes\beta)\circ(\alpha\otimes\mathrm{id}_A) = (\beta\otimes\mathrm{id}_A)\circ(\mathrm{id}_A\otimes\alpha) = (\beta\circ\mathrm{id}_A)\otimes(\mathrm{id}_A\circ\alpha) = \beta\otimes\alpha$$

It requires inserting and removing identities, and using the exchange law in both directions. So, it seems to be very hard to find a generic way to handle formal composites of generators modulo the congruence described above. We will therefore define an alternative construction of these morphisms which doesn't require such a quotienting.

Consider the morphism $\gamma \circ \gamma : (1 \otimes 1) \Rightarrow (1 \otimes 1) : * \to *$ in the theory S of symmetries (Example 1.2), depicted on the left of (2.1):

1 * 1 1_0 $*_0$ 1_1
* γ * $*_1$ γ_0 $*_2$
1 * 1 1_2 $*_3$ 1_3 (2.1)
γ γ_1
1 * 1 1_4 $*_4$ 1_5

Graphically, in this morphism, the two 2-cells are γ, wires are typed by the 1-cell 1 and regions of the plane are typed by the 0-cell $*$. Now, if we give a different name to each *instance* of a generator used in this morphism, for example by numbering them as in the right of (2.1), the morphism itself can be described as the 2-polygraph P defined by $E_0 = \{*_0, \ldots, *_4\}$, $E_1 = \{1_0 : *_1 \to *_0,\ 1_1 : *_0 \to *_2, \ldots,\ 1_5 : *_4 \to *_2\}$ and $E_2 = \{\gamma_0 : 1_0 \otimes 1_1 \Rightarrow 1_2 \otimes 1_3,\ \gamma_1 : 1_2 \otimes 1_3 \Rightarrow 1_4 \otimes 1_5\}$, together with a function ℓ which to every i-generator of this polygraph associates a label, which is an i-generator of S, so that $\ell : P \to S$ is a morphism of polygraphs (ℓ is defined by $\ell(*_i) = *$, $\ell(1_i) = 1$ and $\ell(\gamma_i) = \gamma$). Formulated in categorical terms, (P, ℓ) is an object in the slice category 2-$\mathbf{Pol}{\downarrow}U_2(S)$. Of course, the naming of the instances of the generators occurring in nets is arbitrary, so we have to consider these labeled polygraphs up to bijections, which correspond to injective renaming of instances. Notice that not every such labeled polygraph is the representation of a morphism: we need an inductive construction of those (it seems to be difficult to give a direct characterization of the suitable polygraphs).

Based on these ideas, we describe the category generated by a polygraph S as a category whose cells are polygraphs labeled by S. We suppose fixed a signature 2-polygraph S and write S_i for $U_i(S)$. This is a generalization of the constructions of labeled transition systems, and is reminiscent of pasting schemes [Pow90] and of proof-nets, which is why we call them *polygraphic nets* (or *nets* for short).

The category of 0-*nets* 0-$\mathbf{Net}_{S_0}$ on the 0-polygraph S_0 is the full subcategory of 0-$\mathbf{Pol}{\downarrow}S_0$ whose objects are 0-polygraphs with exactly one 0-cell, labeled by S_0. Concretely, its objects are pairs (n, A), often written A_n, where n is the *name* of the instance (an integer for example) and A an element of $E_0^{S_0}$, called its *label*, and there is a morphism between two objects whenever they have the same label (all those morphisms are invertible). The category of 1-*nets* 1-$\mathbf{Net}_{S_1}$ is the smallest category whose objects are the 0-nets A_i, whose morphisms $(s^f, f, t^f) : A_i \to B_j$ are triples consisting of a 1-polygraph f labeled by S_1 (i.e. an object in 1-$\mathbf{Pol}{\downarrow}S_1$) and two morphisms of labeled polygraphs $s^f : A_i \to f$ and $t^f : B_j \to f$, called *source* and *target*, which are either a 1-polygraph f such that $E_0^f = \{A_i, B_j\}$ and E_1^f contains only one 1-cell $n \in \mathbb{N}$ with A_i as source and B_j as target (and the obvious injections for s^f and t^f), or $A_i = B_j$, $f = A_i$ and $s^f = t^f = \mathrm{id}_{A_i}$ (this is the identity on A_i), or a composite $f \otimes g : A_i \to B_j$ of two morphisms $f : A_i \to C_k$ and $g : C_k \to B_j$. Here, the composite of two such morphisms is defined as the pushout of the diagram $f \xleftarrow{t^f} C_k \xrightarrow{s^g} g$, that is the disjoint union of the polygraphs f and g quotiented by a relation identifying the 0-cell in C_k in the two components of the union.

Example 2.1. If S is the polygraph of symmetries, the composite of the two morphisms $f : *_0 \to *_1$ and $g : *_1 \to *_2$ defined by

$$E_0^f = \{*_0,\ *_1\} \quad E_1^f = \{1_0 : *_0 \to *_1\} \quad E_0^g = \{*_0,\ *_1,\ *_2\} \quad E_1^g = \{1_1 : *_1 \to *_0,\ 1_0 : *_0 \to *_2\}$$

is the morphism $h = f \otimes g$ such that

$$E_0^h = \{*_0, \dots, *_3\} \quad \text{and} \quad E_1^h = \{1_0 : *_0 \to *_1,\ 1_1 : *_1 \to *_3,\ 1_2 : *_3 \to *_2\}$$

Graphically,

$$*_0 \xrightarrow{1_0} *_1 \quad \otimes \quad *_1 \xrightarrow{1_1} *_0 \xrightarrow{1_0} *_2 \quad = \quad *_0 \xrightarrow{1_0} *_1 \xrightarrow{1_1} *_3 \xrightarrow{1_2} *_2$$

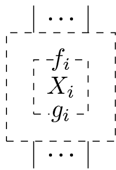
Since composition is defined by a pushout construction, it involves a renaming of some instances (it is the case in the example above) and this renaming is arbitrary. So, composition is not strictly associative but only associative up to isomorphism of polygraphs. Therefore, what we have built is not precisely a category but only a bicategory: this is a well-known fact, this construction being a particular instance of the general construction of cospan bicategories. We can iterate this construction one step further and define the tricategory (that is a 2-category whose compositions are associative up to isomorphism) of 2-*nets* 2-$\mathbf{Net}_S$ as the smallest tricategory whose 0-cells are 0-nets A_i, whose 1-cells $f : A_i \to B_j$ contain 1-nets, and whose 2-cells $\alpha : f \Rightarrow g$ are triples $(s^\alpha, \alpha, t^\alpha)$, consisting of a 2-polygraph α labeled by S and two morphisms of labeled polygraphs $s^\alpha : f \to \alpha$ and $t^\alpha : g \to \alpha$, containing all the 2-polygraphs with one 2-generator $n \in \mathbb{N}$ whose source $f = s_1^\alpha(n)$ and target $g = t_1^\alpha(n)$ are 1-nets which are "disjoint" in the sense they only have their own source and target as common generators, with the obvious injections for s^α and t^α. Moreover, we requires this tricategory to contain identities and to be closed under both vertical and horizontal compositions, which are defined by pushout constructions in a way similar to 1-nets. If we quotient this tricategory and identify cells which are isomorphic labeled polygraphs, we get a proper 2-category, that we still write 2-$\mathbf{Net}_S$.

Proposition 2.2. *The* 2*-category* 2-$\mathbf{Net}_S$ *described above is equivalent to the free category generated by the* 2*-polygraph* S*.*

This construction has the advantage to be simple to implement and manipulate: we have for example given the data needed to describe the morphism (2.1).

3. Critical pairs in polygraphs

In order to formalize the notion of critical pair for a polygraph, we need to formalize first the notion of context of a morphism in the 2-category $\mathcal{C}_2(S)$ generated by a 2-polygraph S, which may be thought as a 2-cell with multiple typed "holes". These contexts have multiples "inputs" (one for each hole) and will therefore organize into a multicategory, which is a notion generalizing categories in the sense that morphisms $f : (A_1, \dots, A_n) \to A$ have one output of type A, and *a list* of inputs of type A_i instead of only one input. Composition is also generalized in the sense that we compose such a morphism f with n morphisms f_i with A_i as target, what we write $f \circ (f_1, \dots, f_n)$. Multicategories should moreover have identities $\mathrm{Id}_A : (A) \to A$ and satisfy coherence axioms [Lei04].



Suppose that we are given a signature 2-polygraph S. Suppose moreover that we are given a list of n pairs of parallel 1-cells (f_i, g_i) in the category generated by the 1-polygraph $U_1(S)$. We write $S[X_1 : f_1 \Rightarrow g_1, \dots, X_n : f_n \Rightarrow g_n]$, for the polygraph obtained from S by adding $X_1, \dots, X_n$ as 2-generators, with f_i as the source and g_i as the target of X_i (we suppose that the X_i were not already present in the 2-generators of S). The X_i should be thought as typed variables for 2-cells and we can easily define a notion of *substitution* of a variable $X_i : f_i \Rightarrow g_i$ by a 2-cell $\alpha : f_i \Rightarrow g_i$ in a 2-cell of the 2-category generated by $S[X_1 : f_1 \Rightarrow g_1, \dots, X_n : f_n \Rightarrow g_n]$.

Given a signature S, we build a multicategory $\mathcal{K}(S)$ whose objects are pairs (f,g) of parallel 1-cells in the 2-category generated by S and whose morphisms, called *contexts*, $K : ((f_1,g_1),\ldots,(f_n,g_n)) \to (f,g)$ are the 2-cells $\alpha : f \Rightarrow g$ in the 2-category generated by the polygraph $S[X_1 : f_1 \Rightarrow g_1,\ldots,X_n : f_n \Rightarrow g_n]$, which are *linear* in the sense that each of the variables X_i appears exactly once in the morphism α. Composition in this multicategory is induced by the substitution operation. This multicategory can be canonically equipped with a structure of symmetric multicategory, which essentially means that, for every permutation σ on n elements, the sets of morphisms of type $((f_1,g_1),\ldots,(f_n,g_n)) \to (f,g)$ is isomorphic to the set of morphisms of type $((f_{\sigma(1)},g_{\sigma(1)}),\ldots,(f_{\sigma(n)},g_{\sigma(n)})) \to (f,g)$ in a coherent way. Any 2-cell $\alpha : f \Rightarrow g$ in the 2-category generated by S, can be seen as a nullary context of type $() \to (f,g)$ that we still write α. A concrete and implementable definition of the multicategory $\mathcal{K}(S)$ of contexts of S can be given by adapting the construction of polygraphic nets given in the previous section.

This construction enables us to reformulate usual notions of rewriting theory in our framework as follows. We suppose fixed a rewriting system given by a 3-polygraph R. We write $S = U_2(R)$ for the underlying signature of R and $\mathcal{C}$ for the 2-category it generates.

Definition 3.1. A *unifier* of two 2-cells

$$\alpha_1 : f_1 \Rightarrow g_1 \quad \text{and} \quad \alpha_2 : f_2 \Rightarrow g_2$$

in $\mathcal{C}$ is a pair of cofinal unary contexts

$$K_1 : ((f_1,g_1)) \to (f,g) \quad \text{and} \quad K_2 : ((f_2,g_2)) \to (f,g)$$

such that $K_1 \circ (\alpha_1) = K_2 \circ (\alpha_2)$. A unifier is a *most general unifier* when it is

- *non-trivial*: there exists no binary context $K : ((f_1,g_1),(f_2,g_2)) \to (f,g)$ such that $K_1 = K \circ (\mathrm{Id}_{(f_1,g_1)},\alpha_2)$ and $K_2 = K \circ (\alpha_1,\mathrm{Id}_{(f_2,g_2)})$. Informally, the morphisms α_1 and α_2 should not appear in disjoint positions in the morphism $K_1\circ(\alpha_1) = K_2\circ(\alpha_2)$.
- *minimal*: for every unifier K_1', K_2' of α_1 and α_2, such that $K_1 = K_1'' \circ K_1'$ and $K_2 = K_2'' \circ K_2'$, for some contexts K_1'' and K_2'', the contexts K_1'' and K_2'' should be invertible.

Remark 3.2. If we write $\alpha = K_1 \circ (\alpha_1) = K_2 \circ (\alpha_2)$ and represent the 2-cells α_1, α_2 and α by 2-nets, the fact that α is a unifier of the morphisms means that there exist two injective morphisms of labeled polygraphs $i_1 : \alpha_1 \to \alpha$ and $i_2 : \alpha_2 \to \alpha$, and the non-triviality condition means that there exists at least one 2-generator which is both in the image of i_1 and i_2.

For example, the last two morphisms of (0.3) are both unifiers of the left members of the rules (0.2). By extension, a unifier of two 3-generators $r_1 : \alpha_1 \Rrightarrow \beta_1$ and $r_2 : \alpha_2 \Rrightarrow \beta_2$ of R is a unifier of their sources α_1 and α_2. A *critical pair* (K_1,r_1,K_2,r_2) consists of a pair of 3-generators r_1, r_2 and a most general unifier K_1, K_2 of those.

Remark 3.3. In Definition 3.1, the 2-cell α_1, can be seen as a context $\alpha_1 : () \to (f_1,g_1)$ in $\mathcal{K}(\mathcal{C})$, and similarly for α_2. In fact, the notion of *unifier* can be generalized to any pair of morphisms in the multicategory $\mathcal{K}(\mathcal{C})$.

A 2-cell $\alpha : f \Rightarrow g$ *rewrites* to a 2-cell $\beta : f \Rightarrow g$, by a 3-generator $r : \alpha' \Rrightarrow \beta' : f' \Rightarrow g'$, when there exists a context $K : ((f',g')) \to (f,g)$ such that $\alpha = K \circ \alpha'$ and $\beta = K \circ \beta'$. In this case, we write $\alpha \Rrightarrow^{K,r} \beta$. The rewriting system R is *terminating* when there is no infinite sequence $\alpha_1 \Rrightarrow^{K_1,r_1} \alpha_2 \Rrightarrow^{K_2,r_2} \ldots$. A *peak* is a triple $(\alpha_1,r_1,\alpha,r_2,\alpha_2)$,

where α, α_1 and α_2 are 2-cells and r_1 and r_2 are 3-generators, such that $\alpha \Rrightarrow^{K_1,r_1}$ and $\alpha \Rrightarrow^{K_2,r_2} \alpha_2$. In particular, with the notations of Definition 3.1, every critical pair induces a peak $(K_1 \circ (\beta_1), r_1, K_1 \circ (\alpha_1), r_2, K_2 \circ (\beta_2))$. A peak is *joinable* when there exist a 2-cell β and 3-cells $\rho_1 : \alpha_2 \Rrightarrow \beta$ and $\rho_2 : \alpha_2 \Rrightarrow \beta$. A rewriting system is *locally* confluent if every peak is joinable. Newman's Lemma is valid for 3-polygraphs [Gui09]:

Proposition 3.4. *A terminating rewriting system is confluent if it is locally confluent.*

Moreover, local confluence can be tested using critical pairs:

Proposition 3.5. *A rewriting system is locally confluent if all its critical pairs are joinable.*

So, in order to test whether a terminating polygraphic rewriting system is confluent, it would be tempting to compute all its critical pairs and test whether they are joinable, as in term rewriting systems. However, as explained in the introduction, even a finite polygraphic rewriting system might admit an infinite number of critical pairs. In the next section, we introduce a theoretical setting which allows us to compute a finite number of generating families of critical pairs.

4. An embedding in compact 2-categories

The notion of adjunction in the 2-category **Cat** of categories, functors and natural transformations can be generalized to any 2-category as follows. Suppose that we are given a 2-category $\mathcal{C}$. A 1-cell $f : A \to B$ is *left adjoint* to a 1-cell $g : B \to A$ (or g is *right adjoint* to f) when there exist two 2-cells $\eta : \mathrm{id}_A \Rightarrow f \otimes g$ and $\varepsilon : g \otimes f \Rightarrow \mathrm{id}_B$, called respectively the *unit* and the *counit* of the adjunction and depicted respectively on the left of (4.1), such that $(f \otimes \varepsilon) \circ (\eta \otimes f) = \mathrm{id}_f$ and $(\varepsilon \otimes g) \circ (g \otimes \eta) = \mathrm{id}_g$. These equations are called the *zig-zag laws* because of their graphical representation, given on the right of (4.1):

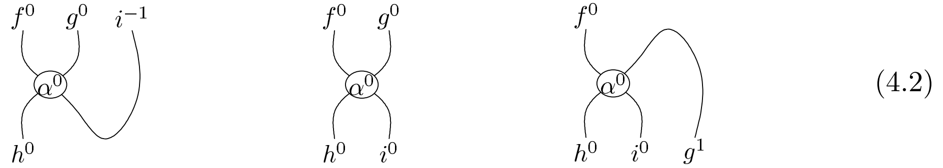
$$f \quad g \qquad g \quad f \qquad f \quad f \;=\; f \quad f \qquad g \quad g \;=\; g \quad g \tag{4.1}$$

A 2-category is *compact* (sometimes also called *autonomous* or *rigid*) when every 1-cell admits both a left and a right adjoint. Given a 2-category $\mathcal{C}$, we write $\overline{\mathcal{C}}$ for the free compact 2-category on $\mathcal{C}$. An explicit description of this 2-category can be given [Kel80]:

- its 0-cells are the 0-cells of $\mathcal{C}$,
- its 1-cells are pairs $f^n : A \to B$ consisting of an integer $n \in \mathbb{Z}$, called *winding number*, and a 1-cell $f : A \to B$ (resp. $f : B \to A$) of $\mathcal{C}$ if n is even (resp. odd),
- a 2-cell is either $\alpha^0 : f^0 \Rightarrow g^0$, where $\alpha : f \Rightarrow g$ is a 2-cell of $\mathcal{C}$, or $\eta_f^n : \mathrm{id}_B \Rightarrow f^n \otimes f^{n+1}$ or $\varepsilon_f^n : f^{n+1} \otimes f^n \Rightarrow \mathrm{id}_A$, where $f^n : A \to B$ is a 1-cell, or a formal vertical or horizontal composite of those,
- 1- and 2-cells are quotiented by a suitable congruence imposing the axioms of 2-categories, compatibility of vertical and horizontal compositions in $\overline{\mathcal{C}}$ with those of $\mathcal{C}$ (for example $(\beta \circ \alpha)^0 = \beta^0 \circ \alpha^0$ and $(\mathrm{id}_f)^0 = \mathrm{id}_{f^0}$) and the zig-zag laws (4.1).

Given a 1-cell f in this category, we often write f^m for the 1-cell defined inductively by $(f \otimes g)^m = f^m \otimes g^m$ and $(f^n)^m = f^{n+m}$ (notice that f^{-1} does not denote the inverse of f in this context). This algebraic construction is important in order to formally define the 2-category $\overline{\mathcal{C}}$ but this construction might be better grasped graphically, with the help of string diagrams: the compact structure adds to $\mathcal{C}$ the possibility to bend wires, without

creating loops. For example, consider a 2-cell $\alpha : f \otimes g \Rightarrow h \otimes i$ in a 2-category $\mathcal{C}$. This 2-cell can be seen as a 2-cell $\alpha^0 : f^0 \otimes g^0 \Rightarrow h^0 \otimes i^0$ of $\overline{\mathcal{C}}$, as pictured in the center of (4.2).



(4.2)

From this morphism, we can deduce a 2-cell $\rho_{f^0,g^0,h^0\otimes i^0}(\alpha) : f^0 \Rightarrow h^0 \otimes i^0 \otimes g^1$, pictured on the right of (4.2), defined by $\rho_{f^0,g^0,h^0\otimes i^0}(\alpha) = (\alpha \otimes \mathrm{id}_{g^1}) \circ (\mathrm{id}_{f^0} \otimes \eta^0_g)$: the wire corresponding to g^0 can be bent on the right and the winding number is increased by one (the output is of type g^1) to "remember" that we have bent the wire once on the right. Similarly, one can define from α the morphism $\rho'_{f^0\otimes g^0,i^0,h^0}(\alpha) : f^0 \otimes g^0 \otimes i^{-1} \Rightarrow h^0$, which corresponds to bending the wire of type i^0 on the left, so its winding number is decreased by 1 (similar transformations can be defined for bending the wires of type f^0 and h^0 in α). Interestingly, by the definition of adjunctions, these two transformations provide mutual inverses: $\rho^{-1}_{f,g,h} = \rho'_{f,g,h}$. We call *rotations* these bijections between the hom-categories of $\overline{\mathcal{C}}$.

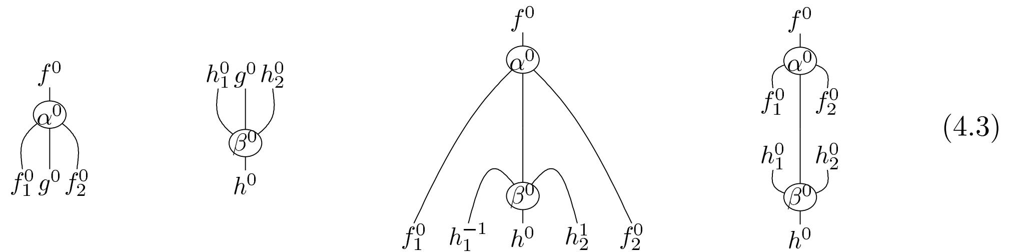
Remark 4.1. The notions of source and target of a 2-cell in a compact 2-category is really artificial since, given a pair of parallel 1-cells $f, g : A \to B$, the rotations induce a bijection between the hom-categories $\mathcal{C}(f, g)$ and $\mathcal{C}(\mathrm{id}_B, f^{-1} \otimes g)$.

It can be shown that the winding numbers on the 1-cells provide enough information about the bending of wires, so that

Proposition 4.2. *Given a 2-category $\mathcal{C}$, the embedding functor $E : \mathcal{C} \to \overline{\mathcal{C}}$ defined as the identity on 0-cells, as $f \mapsto f^0$ on 1-cells and as $\alpha \mapsto \alpha^0$ on 2-cells is full and faithful.*

This means that given two 0-cells A and B of $\mathcal{C}$, the hom-categories $\mathcal{C}(A, B)$ and $\overline{\mathcal{C}}(A, B)$ are isomorphic in a coherent way. The 2-category $\overline{\mathcal{C}}$ thus provides a "larger world" in which we can embed the 2-category $\mathcal{C}$ without losing information.

The interest of this embedding is that there are "extra morphisms" in $\overline{\mathcal{C}}$ that can be used to represent "partial compositions" in $\mathcal{C}$. For example, consider two 2-cells $\alpha : f \Rightarrow f_1 \otimes g \otimes f_2$ and $\beta : h_1 \otimes g \otimes h_2 \Rightarrow h$ in $\mathcal{C}$. These can be seen as the morphisms of $\overline{\mathcal{C}}$ depicted on the left of (4.3) by the previous embedding.



(4.3)

From these two morphisms, the morphism $\alpha \otimes_g \beta : f^0 \Rightarrow f_1^0 \otimes h_1^{-1} \otimes h^0 \otimes h_2^1 \otimes f_2^0$, depicted in the center right of (4.3), can be constructed. This morphism represents the *partial composition* of the 2-cells α and β on the 1-cell g: up to rotations, this 2-cell is fundamentally a way to give a precise meaning to the diagram depicted on the right of (4.3).

The notion of 2-polygraph can easily be adapted to generate compact 2-categories instead of 2-categories. Instead of generating a free category from the underlying 1-polygraph,

we generate a free category with winding numbers: with the notations of Section 1, its objects are the elements of E_0 and its morphisms $f_1^{n_1} \cdot f_2^{n_2} \cdots f_k^{n_k} : A \to B$ are the paths $e(f_1^{n_1}) \cdot e(f_2^{n_2}) \cdots e(f_k^{n_k}) : A \to B$ in the graph described by the 1-polygraph, the edge $e(f^n)$ being f is $n \in \mathbb{Z}$ is even or f taken backwards if f is odd. Similarly, instead of generating a 2-category from the polygraph, we generate a free compact 2-category on the previously generated category with winding numbers with the 2-generators given by the 2-polygraph. Such "polygraphs" are called *compact polygraphs* and we write 2-**CPol** for the category of compact 2-polygraphs. The embedding given in Proposition 4.2 can be extended into an embedding of 2-**Pol** into 2-**CPol**: every 2-polygraph can be seen as a compact 2-polygraph. Given a compact 2-polygraph S, the definition given in Section 3 can be adapted in order to define the multicategory of *compact contexts* $\mathcal{K}(S)$ of S. Finally, the construction of nets given in Section 2 can also be adapted in order to give a concrete and implementable description of the multicategory $\mathcal{K}(S)$ – this essentially amounts to suitably adding winding numbers to 1-cells in the polygraphs involved.

Interestingly, the setting of compact contexts provides a generalization of partial composition by allowing a "partial composition of a morphism with itself". Namely, from a context $\alpha : (\ldots, (f_i, g_i), \ldots) \to (f, g^1 \otimes h \otimes g^0)$ with $f : A \to A$ and $h : B \to B$ one can build the context depicted on the left $\varepsilon_g^0 \circ (g^1 \otimes X \otimes g^0) \circ \alpha : (\ldots, (f_i, g_i), \ldots, (h, \mathrm{id}_B)) \to (f, \mathrm{id}_A)$, where $X : h \to \mathrm{id}_B$ is a fresh variable. This operation amounts to *merging* the outputs of type g^1 and g^0 of α.

5. The unification algorithm

Now that the theoretical setting has been established, we can describe our unification algorithm. Suppose that we are given a polygraphic rewriting system $R \in$ 3-**Pol** whose underlying signature is $S = U_2(R)$. By the previous remarks, S can be seen as a compact 2-polygraph $\overline{S}$. Now, suppose that r_1 and r_2 are two rewriting rules (i.e. 3-generators) in R whose left member are respectively 2-cells $\alpha : f \Rightarrow g$ and $\beta : h \Rightarrow i$. The 2-cell $\alpha : f \Rightarrow g$ in the 2-category generated by S can be seen as a 2-cell $\alpha^0 : f^0 \Rightarrow g^0$ in the compact 2-category $\mathcal{C}$ generated by $\overline{S}$, and therefore as a nullary context $\alpha : () \to (f^0, g^0)$ in the multicategory of contexts $\mathcal{K}(\mathcal{C})$. Similarly, β can be seen as a context $\beta : () \to (h^0, i^0)$. In the multicategory $\mathcal{K}(\mathcal{C})$, we can compute a most general unifier of α and β (see Remark 3.3) from which we will be able to generate critical pairs of the rules r_1 and r_2. Because of space limitations, we don't provide here a fully detailed and formal presentation of the algorithm: the purpose of this paper was to introduce the formal framework necessary to define the algorithm, whose in-depth description will be given in subsequent works.

We first introduce some terminology and notations on nets. Given a 2-net α, an instance of a 2-generator y is the *father* (resp. *son*) of an instance of a 1-generator x if x occurs in the target (resp. source) of y. For example, in (2.1), γ_0 is a son of 1_0 and 1_1 and a father of 1_2 and 1_3. It is easy to show that a given instance of a 1-generator admits at most one father and one son. An instance of 1-generator is *dangling* when it has no father or no son. An instance of a generator is in the *border* of a net if it is in its source or its target.

The algorithm proceeds as follows. We suppose that we have represented the 2-cells α and β as polygraphic 2-nets. Our goal is to construct a 2-net ω together with two injective morphisms of labeled polygraphs $i_1 : \alpha \to \omega$ and $i_2 : \beta \to \omega$ satisfying the properties required for unifiers as reformulated in Remark 3.3. The algorithm is quite similar to the rule-based formulation of the unification algorithm for terms [Baa99]. It begins by

setting $\omega = \alpha$ and $i_1 = \mathrm{id}_\alpha$, and then iterates a procedure that will progressively propagate the unification and make ω grow, by adding cells to it, until it is big enough so that there exists an injection $i_2 : \beta \to \omega$. The procedure which is iterated is non-deterministic and the critical pairs will be obtained as the collection of the results of the non-failed branches of computation. During the iteration two sets are maintained, T and U, which both contains pairs (x, x') consisting of an n-cell x of β and an n-cell x' of ω for some integer $n \in \{0, 1, 2\}$. The set U (for Unified) contains the injection i_2 which is being constructed: if $(x, x') \in U$ and the branch succeeds then the resulting map $i_2 : \beta \to \omega$ will be such that $i_2(x) = x'$. The set T (as in Todo) contains the pairs (x, x') such that x is a cell of β which is to be unified with the cell x' of ω.

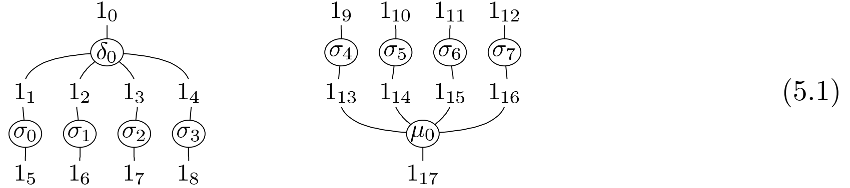
Initially, $\omega = \alpha$, $U = \emptyset$ and $T = \{(x, x')\}$, where x and x' are instances of 2-generators in β and in ω respectively, both chosen non-deterministically. Then the algorithm iterates over the following rules, updating the values of ω, U and T by executing the first rule which applies (updating a value is denoted with the symbol :=).

- *Duplicate.* If $T = \{(x, x')\} \uplus T'$ with $(x, x') \in U$ then $T := T'$.
- *Clash.* If $(x, x') \in T$ and $(x, x'') \in U$ and $x' \neq x''$ then fail.
- *Typecheck.* If $(x, x') \in T$ with $\ell(x) \neq \ell(x')$ then fail.
- *Propagate-0.* If $T = \{(x, x')\} \uplus T'$, where x and x' are 0-cells then
  $T := T'$ and $U := \{(x, x')\} \cup U$.
- *Propagate-1.* If $T = \{(x, x')\} \uplus T'$, where x and x' are 1-cells, then
  $T := T'$ and
  if x has a father y then
    if x' has a father y' then
      $T := \{(y, y')\} \cup T$ and $U := \{(x, x')\} \cup U$
    else either
        add a fresh generator y' of type $\ell(y)$ in ω,
        $T := \{(y, y')\} \cup T$ and $U := \{(x, x')\} \cup U$
      or
        merge x' with some other 1-cell x'' in the border of ω in ω,
        $T := \{(x, x')\} \cup T$
  if x has a son y then
    similar to the previous case.
- *Propagate-2.* If $T = \{(x, x')\} \uplus T'$, where x and x' are 2-cells, then
  $T := T'$, $U := \{(x, x')\} \cup U$, we add in T that the 0- and 1-cells in the source of x should be matched with the corresponding cells in the source of x', and the 0- and 1-cells of the target of x should be matched with those in the target of x'.

The "either... or" construction above denotes a non-deterministic choice and the "merge" refers to the merging operation introduced in Section 4 (this operation might fail if the labels or the winding numbers of x' and x'' are not suitable).

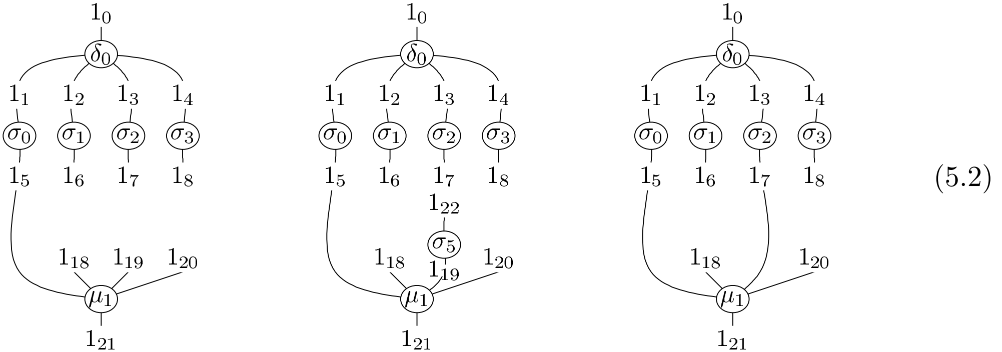
The way this algorithm works is maybe best understood with an example. Consider the signature S with one 0-cell $*$, one 1-cell $1 : * \to *$ and three 2-cells $\delta : 1 \to 4$, $\mu : 4 \to 1$ and $\sigma : 1 \to 1$ (where 4 denotes $1 \otimes 1 \otimes 1 \otimes 1$). We write $\varsigma = \sigma \otimes \sigma \otimes \sigma \otimes \sigma$. Now, consider a rewriting system on this signature containing two rules r_1 and r_2 whose left members are

respectively $\alpha = \varsigma \circ \delta$ and $\beta = \mu \circ \varsigma$, that we represent respectively as the compact nets



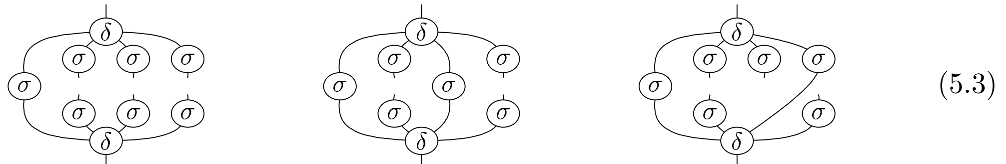
(5.1)

(for simplicity, we omitted the instances of 0-cells). We describe here a few possible non-deterministic branches of the execution of the algorithm. For example, if we begin with $T = \{(\sigma_4, \delta_0)\}$, the algorithm will immediately fail by Typecheck because the label σ of σ_4 differs from the label δ of δ_0. Consider another execution beginning with $T = \{(\sigma_4, \sigma_0)\}$, this time the label matches so Propagate-2 will propagate the unification by setting $T = \{(1_9, 1_1), (1_{13}, 1_5)\}$ and $U = \{(\sigma_4, \sigma_0)\}$. Since 1_9 is dangling, Propagate-1 will move the pair $(1_9, 1_1)$ from T to U. Then the pair $(1_{13}, 1_5)$ will be handled by Propagate-1. Since 1_5 is dangling but 1_{13} is not, a new generator μ_1 will be added to ω (now pictured on the left of (5.2)) and after a few propagations $(1_{13}, 1_5)$ will be moved from T to U, (μ_0, μ_1) will be added to U and T will contain $(1_{11}, 1_{19})$. By Propagate-1, this unification pair can lead to multiple non-deterministic executions: a new generator σ_5 can be added (in the middle of (5.2)), or the 1-generator 1_{19} can be merged with another 1-generator (1_7 for example as pictured in the right of (5.2)). Notice that in this last case, the morphism contains a "hole" of type $1_6 \Rightarrow 1_{18}$, which is handled by a context variable.



(5.2)

By executing fully the algorithm, the three morphisms of (5.3) will be obtained as unifiers (as well as many others).



(5.3)

It can be shown that the algorithm terminates and generates all the critical pairs in compact contexts, and these are in finite number. It is important to notice that the algorithm generates the critical pairs of a rewriting system R in the "bigger world" of compact contexts, from which we can generate the critical pairs in the 2-category generated by R (which are not necessarily in finite number as explained in the introduction). If joinability of the critical pairs in compact contexts implies that the rewriting system is confluent, the converse is unfortunately not true: a similar situation is well known in the

study of λ-calculus with explicit substitution, where a rewriting system might be confluent without being confluent on terms with metavariables.

We have realized a toy implementation of the algorithm in less than 2000 lines of OCaml, with which we have been able to successfully recover the critical pairs of rewriting systems in [Laf03]. Even though we did not particularly focus on efficiency, the execution times are good, typically less than a second, because the morphisms involved in polygraphic rewriting systems are usually small (but they can generate a large number of critical pairs)

Future works. This paper lays the theoretical foundations for unification in polygraphic 2-dimensional rewriting systems and leaves many research tracks open for future works. We plan to study the precise links between our algorithm and the usual unification for terms (every term rewriting system can be seen as a polygraphic rewriting system [Bur93]) as well as algorithms for (planar) graph rewriting. Concerning concrete applications, since these rewriting systems essentially transform circuits made of operators (the 2-generators) linked by a bunch of wires (the 1-generators), it would be interesting to see if these methods can be used to optimize electronic circuits. Finally, we plan investigating the generalization of these methods in dimension higher than 2, which seems to be very challenging.

Acknowledgements. The author is much indebted to John Baez, Albert Burroni, Jonas Frey, Emmanuel Haucourt, Martin Hyland, Yves Lafont, Paul-André Melliès and François Métayer.

References

[Baa99] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.

[Bur93] A. Burroni. Higher-dimensional word problems with applications to equational logic. *Theor. Comput. Sci.*, 115(1):43–62, 1993.

[Gui06a] Y. Guiraud. The three dimensions of proofs. *Ann. pure appl. logic*, 141(1-2):266–295, 2006.

[Gui06b] Y. Guiraud. Two polygraphic presentations of Petri nets. *TCS*, 360(1-3):124–146, 2006.

[Gui09] Y. Guiraud and P. Malbos. Higher-dimensional categories with finite derivation type. *Theor. Appl. Cat.*, 22(18):420–478, 2009.

[Joy91] A. Joyal and R. Street. The Geometry of Tensor Calculus, I. *Adv. Math.*, 88:55–113, 1991.

[Kel80] G.M. Kelly and M.L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

[Laf03] Y. Lafont. Towards an algebraic theory of Boolean circuits. *J. Pure Appl. Alg.*, 184:257–310, 2003.

[Law63] F. W. Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the context of Functorial Semantics of Algebraic Theories.* Ph.D. thesis, Columbia University, 1963.

[Lei04] T. Leinster. *Higher Operads, Higher Categories.* Cambridge University Press, 2004.

[Mac71] S. MacLane. *Categories for the Working Mathematician.* Springer Verlag, 1971.

[Mim08] S. Mimram. *Sémantique des jeux asynchrones et réécriture 2-dimensionnelle.* Ph.D. thesis, 2008.

[Mim09a] S. Mimram. Computing Critical Pairs in Polygraphs, 2009. Preprint.

[Mim09b] S. Mimram. The Structure of First-Order Causality. In *LICS'09*, pp. 212–221. 2009.

[Pow90] J. Power. An n-categorical pasting theorem. *Proc. Int. Conf. Como*, pp. 326–358, 1990.

[Str76] R. Street. Limits indexed by category-valued 2-functors. *J. Pure Appl. Alg.*, 8(2):149–181, 1976.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 243-258
http://rewriting.loria.fr/rta/

POLYNOMIAL INTERPRETATIONS OVER THE REALS DO NOT SUBSUME POLYNOMIAL INTERPRETATIONS OVER THE INTEGERS

FRIEDRICH NEURAUTER AND AART MIDDELDORP

Institute of Computer Science
University of Innsbruck, Austria
E-mail address: {friedrich.neurauter,aart.middeldorp}@uibk.ac.at

ABSTRACT. Polynomial interpretations are a useful technique for proving termination of term rewrite systems. They come in various flavors: polynomial interpretations with real, rational and integer coefficients. In 2006, Lucas proved that there are rewrite systems that can be shown polynomially terminating by polynomial interpretations with real (algebraic) coefficients, but cannot be shown polynomially terminating using polynomials with rational coefficients only. He also proved a similar theorem with respect to the use of rational coefficients versus integer coefficients. In this paper we show that polynomial interpretations with real or rational coefficients do not subsume polynomial interpretations with integer coefficients, contrary to what is commonly believed. We further show that polynomial interpretations with real coefficients subsume polynomial interpretations with rational coefficients.

1. Introduction

Polynomial interpretations are a simple yet useful technique for proving termination of term rewrite systems (TRSs, for short). While originally conceived in the late seventies by Lankford [Lan79] as a means for establishing direct termination proofs, polynomial interpretations are nowadays often used in the context of the dependency pair (DP) framework [Art00, Gie05, Hir05]. In the classical approach of Lankford, one considers polynomials with integer coefficients inducing polynomial algebras over the well-founded domain of the natural numbers. To be precise, every n-ary function symbol f is interpreted by a polynomial P_f in n indeterminates with integer coefficients, which induces a mapping or *interpretation* from terms to integer numbers in the obvious way. In order to conclude termination of a given TRS, three conditions have to be satisfied. First, every polynomial must be *well-defined*, i.e., it must induce a well-defined polynomial function $f_{\mathbb{N}}\colon \mathbb{N}^n \to \mathbb{N}$ over the natural numbers. In addition, the interpretation functions $f_{\mathbb{N}}$ are required to be *strictly monotone* in all arguments. Finally, one has to show *compatibility* of the interpretation with the given TRS. More precisely, for every rewrite rule $l \to r$ the polynomial P_l associated with the left-hand side must be greater than P_r, the corresponding polynomial of the right-hand side, i.e., $P_l > P_r$ for all values of the indeterminates.

1998 ACM Subject Classification: F.4.2 Grammars and Other Rewriting Systems, F.4.1 Mathematical Logic: Computational logic.

Key words and phrases: term rewriting, termination, polynomial interpretations.

© F. Neurauter and A. Middeldorp
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.243

Already back in the seventies, an alternative approach using polynomials with real coefficients instead of integers was proposed by Dershowitz [Der79]. However, as the real numbers $\mathbb{R}$ equipped with the standard order $>_{\mathbb{R}}$ are not well-founded, a subterm property is explicitly required to ensure well-foundedness. And it was not until 2005 that this limitation was overcome, when Lucas [Luc05] presented a framework for proving polynomial termination over the real numbers, where well-foundedness is basically achieved by replacing $>_{\mathbb{R}}$ with a new ordering $>_{\mathbb{R},\delta}$ requiring comparisons between terms to not be below a given positive real number δ. Moreover, this framework also facilitates polynomial interpretations over the rational numbers.

Thus, one can distinguish three variants of polynomial interpretations, polynomial interpretations with real, rational and integer coefficients, and the obvious question is: what is their relationship with regard to termination proving power? For Knuth-Bendix orders it is known [Kor03, Lep01] that extending the range of the underlying weight function from natural numbers to non-negative reals does not result in an increase in termination proving power. In 2006, a partial answer to this question was given by Lucas [Luc06], who managed to show that there are rewrite systems that can be shown polynomially terminating by polynomial interpretations with rational coefficients, but cannot be shown polynomially terminating using polynomials with integer coefficients only. Likewise, he proved that there are systems that can be handled by polynomial interpretations with real (algebraic) coefficients, but cannot be handled by polynomial interpretations with rational coefficients. Based on these results and the fact that we have the strict inclusions $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$, there is the common yet unproven belief in the term rewriting community that polynomial interpretations with real coefficients properly subsume polynomial interpretations with rational coefficients, which in turn properly subsume polynomial interpretations with integer coefficients.[1] However, in this paper we show that it is not true by (constructively) proving that polynomial interpretations with real or rational coefficients do not properly subsume polynomial interpretations with integer coefficients. Besides, we also prove that polynomial interpretations with real coefficients subsume polynomial interpretations with rational coefficients.

The remainder of this paper is organized as follows. In Section 2, we introduce some preliminary definitions and terminology concerning polynomials and polynomial interpretations. In Section 3, we show that polynomial interpretations with real coefficients subsume polynomial interpretations with rational coefficients. Section 4 is dedicated to our main result showing that polynomial interpretations with real or rational coefficients do not properly subsume polynomial interpretations with integer coefficients. We conclude in Section 5.

2. Preliminaries

As usual, we denote by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the sets of natural, integer, rational and real numbers, respectively. An *irrational* number is a real number, which is not in $\mathbb{Q}$. Given some $N \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ and $m \in N$, $>_N$ denotes the standard order of the respective domain and $N_m := \{x \in N \mid x \geq m\}$. A sequence of real numbers $(x_n)_{n \in \mathbb{N}}$ *converges* to the *limit* x if for every real number $\varepsilon > 0$ there exists a natural number N such that the absolute distance $|x_n - x|$ is less than ε for all $n > N$; we denote this by $\lim_{n\to\infty} x_n = x$. As

[1] E.g., [Luc07] states that "polynomial interpretations over the reals are strictly better for proving polynomial termination of rewriting than those which only use integer coefficients".

convergence in $\mathbb{R}^k$ is equivalent to componentwise convergence, we use the same notation also for limits of converging sequences of vectors of real numbers $(\vec{x}_n \in \mathbb{R}^k)_{n\in\mathbb{N}}$. A real function $f\colon \mathbb{R}^k \to \mathbb{R}$ is *continuous* in $\mathbb{R}^k$ if for every converging sequence $(\vec{x}_n \in \mathbb{R}^k)_{n\in\mathbb{N}}$ it holds that $\lim_{n\to\infty} f(\vec{x}_n) = f(\lim_{n\to\infty} \vec{x}_n)$. Finally, as $\mathbb{Q}$ is dense in $\mathbb{R}$, every real number is a rational number or the limit of a converging sequence of rational numbers.

Polynomials

For any ring R (e.g., $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$), we denote the associated *polynomial ring* in n *indeterminates* $x_1,\ldots,x_n$ by $R[x_1,\ldots,x_n]$, the elements of which are finite sums of *monomials* of the form $c \cdot x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$, where the *coefficient* c is an element of R and the *exponents* $i_1,\ldots,i_n$ are natural numbers. An element $P \in R[x_1,\ldots,x_n]$ is called an *(n-variate) polynomial with coefficients in R*. For example, the polynomial $2x^2 - x + 1$ is an element of $\mathbb{Z}[x]$, the ring of all univariate polynomials with integer coefficients. The *degree* of a monomial $c \cdot x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$ is just the sum of its exponents.

In the special case $n = 1$, a polynomial $P \in R[x]$ can be written as follows: $P(x) = \sum_{k=0}^{d} a_k x^k$ $(d \geq 0)$. For the largest k such that $a_k \neq 0$, we call $a_k x^k$ the *leading monomial* of P, a_k its *leading coefficient* and k its *degree*, which is denoted by $\deg(P) = k$. A polynomial $P \in R[x]$ is said to be *linear* if $\deg(P) = 1$, and *quadratic* if $\deg(P) = 2$.

Polynomial Interpretations

We assume familiarity with the basics of term rewriting and polynomial interpretations (e.g. [Baa98, Ter03]). The key concept for establishing (direct) termination of TRSs via polynomial interpretations is the notion of well-founded monotone algebras as they induce reduction orders on terms.

Definition 2.1. Let $\mathcal{F}$ be a signature, i.e., a set of function symbols equipped with fixed arities. A *(well-founded) monotone $\mathcal{F}$-algebra* $(\mathcal{A}, >_A)$ is a non-empty algebra $\mathcal{A} = (A, \{f_A\}_{f\in\mathcal{F}})$ together with a (well-founded) order $>_A$ on the *carrier* A of $\mathcal{A}$ such that every algebra operation f_A is strictly monotone in all arguments, i.e., if $f \in \mathcal{F}$ has arity $n \geq 1$ then $f_A(a_1,\ldots,a_i,\ldots,a_n) >_A f_A(a_1,\ldots,b,\ldots,a_n)$ for all $a_1,\ldots,a_n, b \in A$ and $i \in \{1,\ldots,n\}$ with $a_i >_A b$. Moreover, every function symbol $f \in \mathcal{F}$ is said to be *interpreted* by its associated *interpretation function* f_A.

Given some monotone algebra $(\mathcal{A}, >_A)$, we define the relations $\succeq_\mathcal{A}$ and $\succ_\mathcal{A}$ on terms as follows: $s \succeq_\mathcal{A} t$ if $[\alpha]_\mathcal{A}(s) \geq_A [\alpha]_\mathcal{A}(t)$ and $s \succ_\mathcal{A} t$ if $[\alpha]_\mathcal{A}(s) >_A [\alpha]_\mathcal{A}(t)$, for all assignments α of elements of A to the variables in s and t ($[\alpha]_\mathcal{A}(\cdot)$ denotes the usual evaluation function associated with the algebra $\mathcal{A}$). Now if $(\mathcal{A}, >_A)$ is a well-founded monotone algebra, then $\succ_\mathcal{A}$ is a reduction order that can be used to prove termination of TRSs via the following theorem.

Theorem 2.2. *A TRS is terminating if and only if it is compatible with a well-founded monotone algebra.*

Here, a TRS $\mathcal{R}$ is *compatible* with a well-founded monotone algebra $(\mathcal{A}, >_A)$ if $l \succ_\mathcal{A} r$ for every rewrite rule $l \to r \in \mathcal{R}$.

Definition 2.3. A *polynomial interpretation over* $\mathbb{N}$ for a signature $\mathcal{F}$ consists of a polynomial $f_{\mathbb{N}} \in \mathbb{Z}[x_1, \ldots, x_n]$ for every n-ary function symbol $f \in \mathcal{F}$, such that for all $f \in \mathcal{F}$ the following two properties are satisfied:

(1) *well-definedness*: $f_{\mathbb{N}}(x_1, \ldots, x_n) \in \mathbb{N}$ for all $x_1, \ldots, x_n \in \mathbb{N}$,
(2) *strict monotonicity* of $f_{\mathbb{N}}$ in all arguments with respect to $>_{\mathbb{N}}$, the standard order on $\mathbb{N}$.

Now $(\mathbb{N}, \{f_{\mathbb{N}}\}_{f \in \mathcal{F}}, >_{\mathbb{N}})$ constitutes a well-founded monotone algebra, and we say that a polynomial interpretation over $\mathbb{N}$ is *compatible* with a TRS $\mathcal{R}$ if the well-founded monotone algebra $(\mathbb{N}, \{f_{\mathbb{N}}\}_{f \in \mathcal{F}}, >_{\mathbb{N}})$ is compatible with $\mathcal{R}$. Finally, a TRS is *polynomially terminating over* $\mathbb{N}$ if it admits a compatible polynomial interpretation over $\mathbb{N}$.

Remark 2.4. In principle, one could take any set $\mathbb{N}_m$ (or even $\mathbb{Z}_m$) instead of $\mathbb{N}$ as the carrier for polynomial interpretations. However, it is well-known [Ter03, Con05] that all these sets are order-isomorphic to $\mathbb{N}$ and hence do not change the class of polynomially terminating TRSs. In other words, a TRS $\mathcal{R}$ is polynomially terminating over $\mathbb{N}$ if and only if it is polynomially terminating over $\mathbb{N}_m$. Thus, we can restrict to $\mathbb{N}$ as carrier without loss of generality.

Now if one wants to extend the notion of polynomial interpretations to the rational or real numbers, the main problem one is confronted with is the non-well-foundedness of these domains with respect to the standard orders $>_{\mathbb{Q}}$ and $>_{\mathbb{R}}$. In [Hof01, Luc05], this problem is overcome by replacing these orders with new non-total orders $>_{\mathbb{R},\delta}$ and $>_{\mathbb{Q},\delta}$, the first of which is defined as follows: given some fixed positive real number δ,

$$x >_{\mathbb{R},\delta} y \quad :\Longleftrightarrow \quad x - y \geq_{\mathbb{R}} \delta \quad \text{for all } x, y \in \mathbb{R}.$$

Analogously, one defines $>_{\mathbb{Q},\delta}$ on $\mathbb{Q}$. Thus, $>_{\mathbb{R},\delta}$ ($>_{\mathbb{Q},\delta}$) is well-founded on subsets of $\mathbb{R}$ ($\mathbb{Q}$) that are bounded from below. Therefore, any set $\mathbb{R}_m$ ($\mathbb{Q}_m$) could be used as carrier for polynomial interpretations over $\mathbb{R}$ ($\mathbb{Q}$). However, without loss of generality we may restrict to $\mathbb{R}_0$ ($\mathbb{Q}_0$) because the main argument of Remark 2.4 also applies to polynomials over $\mathbb{R}$ ($\mathbb{Q}$), as is already mentioned in [Luc05].

Definition 2.5. A *polynomial interpretation over* $\mathbb{R}$ for a signature $\mathcal{F}$ consists of a polynomial $f_{\mathbb{R}} \in \mathbb{R}[x_1, \ldots, x_n]$ for every n-ary function symbol $f \in \mathcal{F}$ and some positive real number $\delta >_{\mathbb{R}} 0$, such that for all $f \in \mathcal{F}$:

(a) *well-definedness*: $f_{\mathbb{R}}(x_1, \ldots, x_n) \in \mathbb{R}_0$ for all $x_1, \ldots, x_n \in \mathbb{R}_0$
(b) *strict monotonicity* of $f_{\mathbb{R}}$ in all arguments with respect to $>_{\mathbb{R}_0,\delta}$, the restriction of $>_{\mathbb{R},\delta}$ to $\mathbb{R}_0$.

Analogously, one defines polynomial interpretations over $\mathbb{Q}$ by the obvious adaptation of the definition above. Again, $(\mathbb{R}_0, \{f_{\mathbb{R}}\}_{f \in \mathcal{F}}, >_{\mathbb{R}_0,\delta})$ and $(\mathbb{Q}_0, \{f_{\mathbb{Q}}\}_{f \in \mathcal{F}}, >_{\mathbb{Q}_0,\delta})$ constitute well-founded monotone algebras, and we say that a TRS is *polynomially terminating over* $\mathbb{R}$ ($\mathbb{Q}$) if it is compatible with such an algebra.

We conclude this section with a more useful characterization of monotonicity with respect to the orders $>_{\mathbb{R}_0,\delta}$ and $>_{\mathbb{Q}_0,\delta}$ than the one obtained by specializing Definition 2.1. To this end, we note that a function $f\colon \mathbb{R}_0^n \to \mathbb{R}_0$ is strictly monotone in its i-th argument with respect to $>_{\mathbb{R}_0,\delta}$ if and only if $f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n) \geq_{\mathbb{R}} \delta$ for all $x_1, \ldots, x_n, h \in \mathbb{R}_0$ with $h \geq_{\mathbb{R}} \delta$. From this and from the analogous characterization of $>_{\mathbb{Q}_0,\delta}$-monotonicity, it is easy to derive the following lemmata.

Lemma 2.6. *A linear polynomial* $f_{\mathbb{R}}(x_1,\ldots,x_n) = \sum_{i=1}^{n} a_i x_i + a_0$ *in* $\mathbb{R}[x_1,\ldots,x_n]$ *is strictly monotone in all arguments with respect to* $>_{\mathbb{R}_0,\delta}$ *if and only if* $a_i \geq_{\mathbb{R}} 1$ *for all* $i \in \{1,\ldots,n\}$.

Lemma 2.7. *A linear polynomial* $f_{\mathbb{Q}}(x_1,\ldots,x_n) = \sum_{i=1}^{n} a_i x_i + a_0$ *in* $\mathbb{Q}[x_1,\ldots,x_n]$ *is strictly monotone in all arguments with respect to* $>_{\mathbb{Q}_0,\delta}$ *if and only if* $a_i \geq_{\mathbb{Q}} 1$ *for all* $i \in \{1,\ldots,n\}$.

In the remainder of this paper we will sometimes use the term "polynomial interpretations with integer coefficients" as a synonym for polynomial interpretations over $\mathbb{N}$. Likewise, the term "polynomial interpretations with real (rational) coefficients" refers to polynomial interpretations over $\mathbb{R}$ ($\mathbb{Q}$).

3. Polynomial Termination over the Reals and Rationals

In this section we show that polynomial termination over $\mathbb{Q}$ implies polynomial termination over $\mathbb{R}$. The proof is based upon the fact that polynomials induce continuous functions, whose behavior at irrational points is completely defined by the values they take at rational points.

Lemma 3.1. *Let* $f\colon \mathbb{R}^k \to \mathbb{R}$ *be continuous in* $\mathbb{R}^k$. *If* $f(x_1,\ldots,x_k) \geq 0$ *for all* $x_1,\ldots,x_k \in \mathbb{Q}_0$ *then* $f(x_1,\ldots,x_k) \geq 0$ *for all* $x_1,\ldots,x_k \in \mathbb{R}_0$.

Proof. Let $\vec{x} := (x_1,\ldots,x_k) \in \mathbb{R}_0^k$ and let $(\vec{x}_n)_{n\in\mathbb{N}}$ be a sequence of vectors of non-negative rational numbers $\vec{x}_n \in \mathbb{Q}_0^k$ whose limit is $\vec{x}$. Such a sequence exists because $\mathbb{Q}^k$ is dense in $\mathbb{R}^k$. Then

$$f(\vec{x}) = f(\lim_{n\to\infty} \vec{x}_n) = \lim_{n\to\infty} f(\vec{x}_n)$$

by continuity of f. Thus $f(\vec{x})$ is the limit of $(f(\vec{x}_n))_{n\in\mathbb{N}}$, which is a sequence of non-negative real numbers by assumption. Hence, $f(\vec{x})$ is non-negative, too. ∎

Theorem 3.2. *If a TRS is polynomially terminating over* $\mathbb{Q}$, *then it is also polynomially terminating over* $\mathbb{R}$.

Proof. Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$ that is polynomially terminating over $\mathbb{Q}$. So there exists some polynomial interpretation $\mathcal{I}$ over $\mathbb{Q}$ consisting of a positive rational number δ and a polynomial $f_{\mathbb{Q}} \in \mathbb{Q}[x_1,\ldots,x_n]$ for every n-ary function symbol $f \in \mathcal{F}$ such that:

(a) for all n-ary $f \in \mathcal{F}$, $f_{\mathbb{Q}}(x_1,\ldots,x_n) \geq 0$ for all $x_1,\ldots,x_n \in \mathbb{Q}_0$,
(b) for all $f \in \mathcal{F}$, $f_{\mathbb{Q}}$ is strictly monotone with respect to $>_{\mathbb{Q}_0,\delta}$ in all arguments,
(c) for every rewrite rule $l \to r \in \mathcal{R}$, $P_l >_{\mathbb{Q}_0,\delta} P_r$ for all $x_1,\ldots,x_m \in \mathbb{Q}_0$.

Here P_l (P_r) denotes the polynomial associated with l (r) and the variables $x_1,\ldots,x_m$ are those occurring in $l \to r$. Next we note that all three conditions are quantified polynomial inequalities of the shape "$P(x_1,\ldots,x_k) \geq 0$ for all $x_1,\ldots,x_k \in \mathbb{Q}_0$" for some polynomial P with rational coefficients. This is easy to see for the first and third condition. As to the second condition, the function $f_{\mathbb{Q}}$ is strictly monotone in its i-th argument with respect to $>_{\mathbb{Q}_0,\delta}$ if and only if $f_{\mathbb{Q}}(x_1,\ldots,x_i+h,\ldots,x_n) - f_{\mathbb{Q}}(x_1,\ldots,x_i,\ldots,x_n) \geq \delta$ for all $x_1,\ldots,x_n,h \in \mathbb{Q}_0$ with $h \geq \delta$, which is equivalent to

$$f_{\mathbb{Q}}(x_1,\ldots,x_i+\delta+h,\ldots,x_n) - f_{\mathbb{Q}}(x_1,\ldots,x_i,\ldots,x_n) - \delta \geq 0$$

for all $x_1,\ldots,x_n,h \in \mathbb{Q}_0$. From Lemma 3.1 and the fact that polynomials induce continuous functions we infer that all these polynomial inequalities do not only hold in $\mathbb{Q}_0$ but also in $\mathbb{R}_0$. Hence, the polynomial interpretation $\mathcal{I}$ proves termination over $\mathbb{R}$. ∎

We conclude this section with the following remark that emphasizes the essence of the proof of Theorem 3.2.

Remark 3.3. Not only does the result established in this section show that polynomial termination over $\mathbb{Q}$ implies polynomial termination over $\mathbb{R}$, but it even reveals that the same interpretation applies.

4. Polynomial Termination over the Reals and Integers

As far as the relationship of polynomial interpretations with real, rational and integer coefficients with regard to termination proving power is concerned, the only results published to date are due to Lucas [Luc06], who managed to prove the following two theorems.

Theorem 4.1 (Lucas, 2006)**.** *There are TRSs that are polynomially terminating over* $\mathbb{Q}$ *but not over* $\mathbb{N}$*.*

Theorem 4.2 (Lucas, 2006)**.** *There are TRSs that are polynomially terminating over* $\mathbb{R}$ *but not over* $\mathbb{Q}$*.*

Hence, the extension of the coefficient domain from the integers to the rational numbers entails the possibility to prove some rewrite systems polynomially terminating, which could not be proved polynomially terminating otherwise. Moreover, a similar statement holds for the extension of the coefficient domain from the rational numbers to the real numbers. Based on these results and the fact that we have the strict inclusions $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$, it is tempting to believe that polynomial interpretations with real coefficients properly subsume polynomial interpretations with rational coefficients, which in turn properly subsume polynomial interpretations with integer coefficients. Indeed, the former proposition holds according to Theorem 3.2. However, the latter proposition does not hold, as will be shown in this section. In particular, we present a TRS that can be proved terminating by a polynomial interpretation with integer coefficients, but cannot be proved terminating by a polynomial interpretation with real or rational coefficients.

4.1. Motivation

In order to motivate the construction of this particular rewrite system, let us first observe that from the viewpoint of number theory there is a fundamental difference between the integers and the real or rational numbers. More precisely, the integers are an example of a discrete domain, whereas both the real and rational numbers are *dense*[2] domains. In the context of polynomial interpretations, the consequences of this major distinction are best explained by an example. To this end, we consider the polynomial function $x \mapsto 2x^2 - x$ depicted in Figure 1 and assume that we want to use it as the interpretation of some unary function symbol. Now the point is that this function is permissible in a polynomial interpretation over $\mathbb{N}$ as it is both non-negative and strictly monotone over the natural numbers. However, viewing it as a function over a real (rational) variable, we observe that non-negativity is violated in the open interval $(0, \frac{1}{2})$ (and monotonicity requires a properly chosen value for δ). Hence, the polynomial function $x \mapsto 2x^2 - x$ is not permissible in any polynomial interpretation over $\mathbb{R}$ ($\mathbb{Q}$).

[2]Given two distinct real (rational) numbers a and b, there exists a real (rational) number c in between.

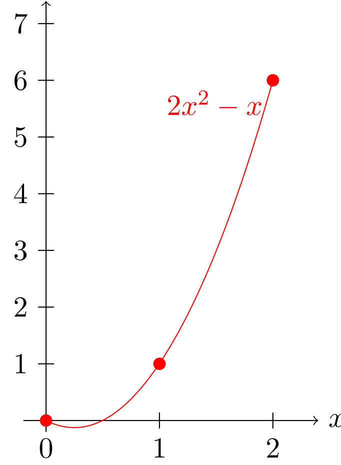


Figure 1: The polynomial function $x \mapsto 2x^2 - x$.

Thus, the idea is to design a rewrite system that enforces an interpretation of this shape for some unary function symbol, and the tool that can be used to achieve this is polynomial interpolation. To this end, let us consider the following scenario, which is fundamentally based on the assumption that some unary function symbol f is interpreted by a quadratic polynomial $\mathsf{f}(x) = ax^2 + bx + c$ with (unknown) coefficients a, b and c. Then, by polynomial interpolation, these coefficients are uniquely determined by the image of f at three pairwise different locations; in this way the interpolation constraints $\mathsf{f}(0) = 0$, $\mathsf{f}(1) = 1$ and $\mathsf{f}(2) = 6$ enforce the interpretation $\mathsf{f}(x) = 2x^2 - x$. Next we encode these constraints in terms of the TRS $\mathcal{R}$ consisting of the following rewrite rules, where $\mathsf{s}^n(x)$ abbreviates $\underbrace{\mathsf{s}(\mathsf{s}(\cdots \mathsf{s}}_{n\text{-times}}(x)\cdots))$,

$$\begin{aligned} \mathsf{s}(\mathsf{0}) &\to \mathsf{f}(\mathsf{0}) & & \\ \mathsf{s}^2(\mathsf{0}) &\to \mathsf{f}(\mathsf{s}(\mathsf{0})) & \mathsf{f}(\mathsf{s}(\mathsf{0})) &\to \mathsf{0} \\ \mathsf{s}^7(\mathsf{0}) &\to \mathsf{f}(\mathsf{s}^2(\mathsf{0})) & \mathsf{f}(\mathsf{s}^2(\mathsf{0})) &\to \mathsf{s}^5(\mathsf{0}) \end{aligned}$$

and consider the following two cases: polynomial interpretations over $\mathbb{N}$ on the one hand and polynomial interpretations over $\mathbb{R}$ on the other hand.

In the context of polynomial interpretations over $\mathbb{N}$, we observe that if we equip the function symbols s and $\mathsf{0}$ with the (natural) interpretations $\mathsf{s}_{\mathbb{N}}(x) = x + 1$ and $\mathsf{0}_{\mathbb{N}} = 0$, then the TRS $\mathcal{R}$ indeed implements the above interpolation constraints.[3] For example, the constraint $\mathsf{f}_{\mathbb{N}}(1) = 1$ is expressed by $\mathsf{f}(\mathsf{s}(\mathsf{0})) \to \mathsf{0}$ and $\mathsf{s}^2(\mathsf{0}) \to \mathsf{f}(\mathsf{s}(\mathsf{0}))$. The former encodes $\mathsf{f}_{\mathbb{N}}(1) > 0$, whereas the latter encodes $\mathsf{f}_{\mathbb{N}}(1) < 2$. Moreover, the rule $\mathsf{s}(\mathsf{0}) \to \mathsf{f}(\mathsf{0})$ encodes $\mathsf{f}_{\mathbb{N}}(0) < 1$, which is equivalent to $\mathsf{f}_{\mathbb{N}}(0) = 0$ in the domain of the natural numbers. Thus, this interpolation constraint can be expressed by a single rewrite rule, whereas the other two constraints require two rules each. Summing up, by virtue of the method of polynomial interpolation, we have reduced the problem of enforcing a specific interpretation for some unary function symbol to the problem of enforcing natural semantics for the symbols s and $\mathsf{0}$.

[3]In fact, one can even show that $\mathsf{s}_{\mathbb{N}}(x) = x + 1$ is sufficient for this purpose.

Next we elaborate on the ramifications of considering the TRS $\mathcal{R}$ in the context of polynomial interpretations over $\mathbb{R}$. To this end, let us assume that the symbols s and $\mathsf{0}$ are interpreted by $\mathsf{s}_{\mathbb{R}}(x) = x + s_0$ and $\mathsf{0}_{\mathbb{R}} = 0$, so that s has some kind of successor function semantics. Then the TRS $\mathcal{R}$ translates to the following constraints:

$$s_0 - \delta \geq_{\mathbb{R}} \mathsf{f}_{\mathbb{R}}(0)$$

$$2s_0 - \delta \geq_{\mathbb{R}} \mathsf{f}_{\mathbb{R}}(s_0) \qquad \mathsf{f}_{\mathbb{R}}(s_0) \geq_{\mathbb{R}} 0 + \delta$$

$$7s_0 - \delta \geq_{\mathbb{R}} \mathsf{f}_{\mathbb{R}}(2s_0) \qquad \mathsf{f}_{\mathbb{R}}(2s_0) \geq_{\mathbb{R}} 5s_0 + \delta$$

Hence, $\mathsf{f}_{\mathbb{R}}(0)$ is confined to the closed interval $[0, s_0 - \delta]$, whereas $\mathsf{f}_{\mathbb{R}}(s_0)$ is confined to $[0 + \delta, 2s_0 - \delta]$ and $\mathsf{f}_{\mathbb{R}}(2s_0)$ to $[5s_0 + \delta, 7s_0 - \delta]$. Basically, this means that these constraints do not uniquely determine the function $\mathsf{f}_{\mathbb{R}}$. In other words, the method of polynomial interpolation does not readily apply to the case of polynomial interpretations over $\mathbb{R}$. However, we can make it work. To this end, we observe that if $s_0 = \delta$, then the above system of inequalities actually turns into the following system of equations, which can be viewed as a set of interpolation constraints (parameterized by s_0) that uniquely determine $\mathsf{f}_{\mathbb{R}}$:

$$\mathsf{f}_{\mathbb{R}}(0) = 0 \qquad \mathsf{f}_{\mathbb{R}}(s_0) = s_0 \qquad \mathsf{f}_{\mathbb{R}}(2s_0) = 6s_0$$

Clearly, if $s_0 = \delta = 1$, then the symbol f is fixed to the interpretation $2x^2 - x$, as was the case in the context of polynomial interpretations over $\mathbb{N}$ (note that in the latter case $\delta = 1$ is implicit because of the equivalence $x >_{\mathbb{N}} y \iff x \geq_{\mathbb{N}} y + 1$). Hence, we conclude that once we can manage to design a TRS that enforces $s_0 = \delta$, we can again leverage the method of polynomial interpolation to enforce a specific interpretation for some unary function symbol. Moreover, we remark that the actual value of s_0 is irrelevant for achieving our goal. That is to say that s_0 only serves as a scale factor in the interpolation constraints determining $\mathsf{f}_{\mathbb{R}}$. Clearly, if $s_0 \neq 1$, then $\mathsf{f}_{\mathbb{R}}$ is not fixed to the interpretation $2x^2 - x$, however, it is still fixed to an interpretation of the same (desired) shape. But more on this later.

4.2. Main Theorem

In the previous subsection we have presented the basic method that we use in order to show that polynomial interpretations with real or rational coefficients do not properly subsume polynomial interpretations with integer coefficients. The construction presented there was based on several assumptions, the essential ones of which are:

(a) The symbol s had to be interpreted by a linear polynomial of the shape $x + s_0$.
(b) The condition $s_0 = \delta$ was required to hold.
(c) The function symbol f had to be interpreted by a quadratic polynomial.

Now the point is that one can get rid of all these assumptions by adding suitable rewrite rules to the TRS $\mathcal{R}$. The resulting TRS will be referred to as $\mathcal{S}$, and it consists of the following rewrite rules:

$$\begin{array}{rclr} \mathsf{s}(\mathsf{0}) & \to & \mathsf{f}(\mathsf{0}) & (1) \\ \mathsf{s}^2(\mathsf{0}) & \to & \mathsf{f}(\mathsf{s}(\mathsf{0})) & (2) \\ \mathsf{s}^7(\mathsf{0}) & \to & \mathsf{f}(\mathsf{s}^2(\mathsf{0})) & (3) \\ \mathsf{f}(\mathsf{s}(\mathsf{0})) & \to & \mathsf{0} & (4) \\ \mathsf{f}(\mathsf{s}^2(\mathsf{0})) & \to & \mathsf{s}^5(\mathsf{0}) & (5) \\ \mathsf{f}(\mathsf{s}^2(x)) & \to & \mathsf{h}(\mathsf{f}(x), \mathsf{g}(\mathsf{h}(x,x))) & (6) \end{array} \qquad \begin{array}{rclr} \mathsf{f}(\mathsf{g}(x)) & \to & \mathsf{g}(\mathsf{g}(\mathsf{f}(x))) & (7) \\ \mathsf{g}(\mathsf{s}(x)) & \to & \mathsf{s}(\mathsf{s}(\mathsf{g}(x))) & (8) \\ \mathsf{g}(x) & \to & \mathsf{h}(x,x) & (9) \\ \mathsf{s}(x) & \to & \mathsf{h}(\mathsf{0},x) & (10) \\ \mathsf{s}(x) & \to & \mathsf{h}(x,\mathsf{0}) & (11) \\ \mathsf{h}(\mathsf{f}(x),\mathsf{g}(x)) & \to & \mathsf{f}(\mathsf{s}(x)) & (12) \end{array}$$

In this system the rewrite rules (7) and (8) serve the purpose of ensuring the first of the above items. Informally, (8) constrains the interpretation of the symbol s to a linear polynomial by simple reasoning about the degrees of the left- and right-hand side polynomials, and (7) does the same thing with respect to g. Because both interpretations are linear, compatibility with (8) can only be achieved if the leading coefficient of the interpretation of s is one.

Concerning item (c) above, we remark that the tricky part is to enforce the upper bound of two on the degree of the polynomial $\mathsf{f}_\mathbb{R}$ that interprets the symbol f. To this end, we make the following observation. If $\mathsf{f}_\mathbb{R}$ is at most quadratic, then the function $\mathsf{f}_\mathbb{R}(x+s_0)-\mathsf{f}_\mathbb{R}(x)$ is at most linear; that is, there is a linear function $\mathsf{g}_\mathbb{R}(x)$ such that $\mathsf{g}_\mathbb{R}(x) > \mathsf{f}_\mathbb{R}(x+s_0)-\mathsf{f}_\mathbb{R}(x)$, or equivalently, $\mathsf{f}_\mathbb{R}(x)+\mathsf{g}_\mathbb{R}(x) > \mathsf{f}_\mathbb{R}(x+s_0)$, for all values of x. This can be encoded in terms of rule (12) as soon as the interpretation of h corresponds to addition of two numbers. And this is exactly the purpose of rules (9), (10) and (11). More precisely, by linearity of the interpretation of g, we infer from (9) that the interpretation of h must have the linear shape $h_2x+h_1y+h_0$. Furthermore, compatibility with (10) and (11) implies $h_2=h_1=1$ due to item (a) above. Hence, the interpretation of h is $x+y+h_0$, and it really models addition of two numbers (modulo adding a constant).

Next we comment on how to enforce the second of the above assumptions. To this end, we remark that the hard part is to enforce the condition $s_0 \leq \delta$. The idea is as follows. First, we consider rule (2), observing that if f is interpreted by a quadratic polynomial $\mathsf{f}_\mathbb{R}$ and s by the linear polynomial $x+s_0$, then (the interpretation of) its right-hand side will eventually become larger than its left-hand side with growing s_0, thus violating compatibility. In this way, s_0 is bounded from above, and the faster the growth of $\mathsf{f}_\mathbb{R}$, the lower the bound. The problem with this statement, however, is that it is only true if $\mathsf{f}_\mathbb{R}$ is fixed (which is a priori not the case); otherwise, for any given value of s_0, one can always find a quadratic polynomial $\mathsf{f}_\mathbb{R}$ such that compatibility with (2) is satisfied. The parabolic curve associated with $\mathsf{f}_\mathbb{R}$ only has to be flat enough. So, in order to prevent this, we have to somehow control the growth of $\mathsf{f}_\mathbb{R}$. Now that is where rule (6) comes into play, which basically expresses that if you increase the argument of $\mathsf{f}_\mathbb{R}$ by a certain amount (i.e., $2s_0$), then the value of the function is guaranteed to increase by a certain minimum amount, too. Thus, this rule establishes a lower bound on the growth of $\mathsf{f}_\mathbb{R}$. And it turns out that if $\mathsf{f}_\mathbb{R}$ has just the right amount of growth, then we can readily establish the desired upper bound δ for s_0.

Finally, having presented all the relevant details of our construction, it remains to formally prove our main claim that the TRS $\mathcal{S}$ is polynomially terminating over $\mathbb{N}$, but not over $\mathbb{R}$ or $\mathbb{Q}$.

Lemma 4.3. *The TRS $\mathcal{S}$ is polynomially terminating over $\mathbb{N}$.*

Proof. We consider the following interpretation:

$$\mathsf{0}_\mathbb{N}=0 \qquad \mathsf{s}_\mathbb{N}(x)=x+1 \qquad \mathsf{f}_\mathbb{N}(x)=2x^2-x \qquad \mathsf{g}_\mathbb{N}(x)=4x+4 \qquad \mathsf{h}_\mathbb{N}(x,y)=x+y$$

Note that the polynomial $2x^2 - x$ is a permissible interpretation function as it is both non-negative and strictly monotone over the natural numbers (cf. Figure 1). The rewrite rules of $\mathcal{S}$ are compatible with this interpretation because the resulting inequalities

$$\begin{aligned} 1 &>_{\mathbb{N}} 0 & 32x^2 + 60x + 28 &>_{\mathbb{N}} 32x^2 - 16x + 20 \\ 2 &>_{\mathbb{N}} 1 & 4x + 8 &>_{\mathbb{N}} 4x + 6 \\ 7 &>_{\mathbb{N}} 6 & 4x + 4 &>_{\mathbb{N}} 2x \\ 1 &>_{\mathbb{N}} 0 & x + 1 &>_{\mathbb{N}} x \\ 6 &>_{\mathbb{N}} 5 & x + 1 &>_{\mathbb{N}} x \\ 2x^2 + 7x + 6 &>_{\mathbb{N}} 2x^2 + 7x + 4 & 2x^2 + 3x + 4 &>_{\mathbb{N}} 2x^2 + 3x + 1 \end{aligned}$$

are clearly satisfied for all natural numbers x. ∎

Lemma 4.4. *The TRS $\mathcal{S}$ is not polynomially terminating over $\mathbb{R}$.*

Proof. Let us assume that $\mathcal{S}$ is polynomially terminating over $\mathbb{R}$ and derive a contradiction. Compatibility with rule (8) implies

$$\deg(\mathsf{g}_{\mathbb{R}}(x)) \cdot \deg(\mathsf{s}_{\mathbb{R}}(x)) \geq \deg(\mathsf{s}_{\mathbb{R}}(x)) \cdot \deg(\mathsf{s}_{\mathbb{R}}(x)) \cdot \deg(\mathsf{g}_{\mathbb{R}}(x))$$

As a consequence, $\deg(\mathsf{s}_{\mathbb{R}}(x)) \leq 1$, and because $\mathsf{s}_{\mathbb{R}}$ and $\mathsf{g}_{\mathbb{R}}$ must be strictly monotone, we conclude $\deg(\mathsf{s}_{\mathbb{R}}(x)) = 1$. The same reasoning applied to rule (7) yields $\deg(\mathsf{g}_{\mathbb{R}}(x)) = 1$. Hence, the symbols s and g must be interpreted by linear polynomials. So $\mathsf{s}_{\mathbb{R}}(x) = s_1 x + s_0$ and $\mathsf{g}_{\mathbb{R}}(x) = g_1 x + g_0$ with $s_0, g_0 \in \mathbb{R}_0$ and, due to Lemma 2.6, $s_1 \geq_{\mathbb{R}} 1$ and $g_1 \geq_{\mathbb{R}} 1$. Then the compatibility constraint imposed by rule (8) gives rise to the inequality

$$g_1 s_1 x + g_1 s_0 + g_0 >_{\mathbb{R}_0,\delta} s_1^2 g_1 x + s_1^2 g_0 + s_1 s_0 + s_0 \tag{13}$$

which must hold for all non-negative real numbers x. This implies the following condition on the respective leading coefficients: $g_1 s_1 \geq_{\mathbb{R}} s_1^2 g_1$. Because of $s_1 \geq_{\mathbb{R}} 1$ and $g_1 \geq_{\mathbb{R}} 1$, this can only hold if $s_1 = 1$. Hence, $\mathsf{s}_{\mathbb{R}}(x) = x + s_0$. This result simplifies (13) to $g_1 s_0 >_{\mathbb{R}_0,\delta} 2s_0$, which implies $g_1 s_0 >_{\mathbb{R}} 2s_0$. From this, we conclude that $s_0 >_{\mathbb{R}} 0$ and $g_1 >_{\mathbb{R}} 2$.

Now suppose that the function symbol f were also interpreted by a linear polynomial $\mathsf{f}_{\mathbb{R}}$. Then we could apply the same reasoning to rule (7) because it is structurally equivalent to (8), thus inferring $g_1 = 1$. However, this would contradict $g_1 >_{\mathbb{R}} 2$; therefore, $\mathsf{f}_{\mathbb{R}}$ cannot be linear.

Next we turn our attention to the rewrite rules (9), (10) and (11). Because $\mathsf{g}_{\mathbb{R}}$ is linear, compatibility with (9) constrains the function $h\colon \mathbb{R}_0 \to \mathbb{R}_0, x \mapsto \mathsf{h}_{\mathbb{R}}(x, x)$ to be at most linear. This can only be the case if $\mathsf{h}_{\mathbb{R}}$ contains no monomials of degree two or higher. In other words, $\mathsf{h}_{\mathbb{R}}(x, y) = h_1 \cdot x + h_2 \cdot y + h_0$, where $h_0 \in \mathbb{R}_0$, $h_1 \geq_{\mathbb{R}} 1$ and $h_2 \geq_{\mathbb{R}} 1$ (cf. Lemma 2.6). Because of $\mathsf{s}_{\mathbb{R}}(x) = x + s_0$, compatibility with (11) implies $h_1 = 1$, and compatibility with (10) implies $h_2 = 1$; thus, $\mathsf{h}_{\mathbb{R}}(x, y) = x + y + h_0$.

Using the obtained information in the compatibility constraint associated with rule (12), we get

$$\mathsf{g}_{\mathbb{R}}(x) + h_0 >_{\mathbb{R}_0,\delta} \mathsf{f}_{\mathbb{R}}(x + s_0) - \mathsf{f}_{\mathbb{R}}(x) \quad \text{for all } x \in \mathbb{R}_0$$

This implies that $\deg(\mathsf{g}_{\mathbb{R}}(x) + h_0) \geq \deg(\mathsf{f}_{\mathbb{R}}(x + s_0) - \mathsf{f}_{\mathbb{R}}(x))$, which simplifies to $1 \geq \deg(\mathsf{f}_{\mathbb{R}}(x)) - 1$ because $s_0 \neq 0$. Consequently, $\mathsf{f}_{\mathbb{R}}$ must be a quadratic polynomial. Without loss of generality, let $\mathsf{f}_{\mathbb{R}}(x) = ax^2 + bx + c$, subject to the constraints: $a >_{\mathbb{R}} 0$ and $c \geq_{\mathbb{R}} 0$ because of non-negativity (for all $x \in \mathbb{R}_0$), and $a\delta + b \geq_{\mathbb{R}} 1$ because $\mathsf{f}_{\mathbb{R}}(\delta) >_{\mathbb{R}_0,\delta} \mathsf{f}_{\mathbb{R}}(0)$ due to strict monotonicity of $\mathsf{f}_{\mathbb{R}}$.

Next we consider the compatibility constraint associated with rule (6), from which we deduce an important auxiliary result. After unraveling the definitions of $>_{\mathbb{R}_0,\delta}$ and the interpretation functions, this constraint simplifies to

$$4as_0x + 4as_0^2 + 2bs_0 \geq_{\mathbb{R}} 2g_1x + g_1h_0 + g_0 + h_0 + \delta \quad \text{for all } x \in \mathbb{R}_0,$$

which implies the following condition on the respective leading coefficients: $4as_0 \geq_{\mathbb{R}} 2g_1$; from this and $g_1 >_{\mathbb{R}} 2$, we conclude

$$as_0 >_{\mathbb{R}} 1 \tag{14}$$

and note that $as_0 = \mathsf{f}'_{\mathbb{R}}(s_0/2) - \mathsf{f}'_{\mathbb{R}}(0)$. Hence, as_0 expresses the change of the slopes of the tangents to $\mathsf{f}_{\mathbb{R}}$ at the points $(0, \mathsf{f}_{\mathbb{R}}(0))$ and $(s_0/2, \mathsf{f}_{\mathbb{R}}(s_0/2))$, and thus (14) actually sets a lower bound on the growth of $\mathsf{f}_{\mathbb{R}}$.

Now let us consider the combined compatibility constraint imposed by rule (2) and rule (4), namely $\mathsf{0}_{\mathbb{R}} + 2s_0 >_{\mathbb{R}_0,\delta} \mathsf{f}_{\mathbb{R}}(\mathsf{s}_{\mathbb{R}}(\mathsf{0}_{\mathbb{R}})) >_{\mathbb{R}_0,\delta} \mathsf{0}_{\mathbb{R}}$, which implies $\mathsf{0}_{\mathbb{R}} + 2s_0 \geq_{\mathbb{R}} \mathsf{0}_{\mathbb{R}} + 2\delta$ by definition of $>_{\mathbb{R}_0,\delta}$. Thus, we conclude $s_0 \geq_{\mathbb{R}} \delta$. In fact, we even have $s_0 = \delta$, which can be derived from the compatibility constraint of rule (2) using the conditions $s_0 \geq_{\mathbb{R}} \delta$, $a\delta + b \geq_{\mathbb{R}} 1$ and $as_0 + b \geq_{\mathbb{R}} 1$, the combination of the former two conditions:

$$\begin{array}{rcl}
\mathsf{0}_{\mathbb{R}} + 2s_0 & >_{\mathbb{R}_0,\delta} & \mathsf{f}_{\mathbb{R}}(\mathsf{s}_{\mathbb{R}}(\mathsf{0}_{\mathbb{R}})) \\
\mathsf{0}_{\mathbb{R}} + 2s_0 - \delta & \geq_{\mathbb{R}} & \mathsf{f}_{\mathbb{R}}(\mathsf{s}_{\mathbb{R}}(\mathsf{0}_{\mathbb{R}})) \\
& = & a(\mathsf{0}_{\mathbb{R}} + s_0)^2 + b(\mathsf{0}_{\mathbb{R}} + s_0) + c \\
& = & a\mathsf{0}_{\mathbb{R}}^2 + \mathsf{0}_{\mathbb{R}}(2as_0 + b) + as_0^2 + bs_0 + c \\
& \geq_{\mathbb{R}} & a\mathsf{0}_{\mathbb{R}}^2 + \mathsf{0}_{\mathbb{R}} + as_0^2 + bs_0 + c \\
& \geq_{\mathbb{R}} & \mathsf{0}_{\mathbb{R}} + as_0^2 + bs_0 \\
& \geq_{\mathbb{R}} & \mathsf{0}_{\mathbb{R}} + as_0^2 + (1 - a\delta)s_0 \\
& = & \mathsf{0}_{\mathbb{R}} + as_0(s_0 - \delta) + s_0
\end{array}$$

Hence, $\mathsf{0}_{\mathbb{R}} + 2s_0 - \delta \geq_{\mathbb{R}} \mathsf{0}_{\mathbb{R}} + as_0(s_0 - \delta) + s_0$, or equivalently, $s_0 - \delta \geq_{\mathbb{R}} as_0(s_0 - \delta)$. But because of (14) and $s_0 \geq_{\mathbb{R}} \delta$, this inequality can only be satisfied if:

$$s_0 = \delta \tag{15}$$

This result has immediate consequences concerning the interpretation of the constant $\mathsf{0}$. To this end, we consider the compatibility constraint of rule (10), which simplifies to $s_0 \geq_{\mathbb{R}} \mathsf{0}_{\mathbb{R}} + h_0 + \delta$. Because of (15) and the fact that $\mathsf{0}_{\mathbb{R}}$ and h_0 must be non-negative, we conclude $\mathsf{0}_{\mathbb{R}} = h_0 = 0$.

Moreover, condition (15) is the key to the proof of this lemma. To this end, we consider the compatibility constraints associated with the five rewrite rules (1)–(5):

$$\begin{array}{rl}
s_0 >_{\mathbb{R}_0,s_0} \mathsf{f}_{\mathbb{R}}(0) & \\
2s_0 >_{\mathbb{R}_0,s_0} \mathsf{f}_{\mathbb{R}}(s_0) & \qquad \mathsf{f}_{\mathbb{R}}(s_0) >_{\mathbb{R}_0,s_0} 0 \\
7s_0 >_{\mathbb{R}_0,s_0} \mathsf{f}_{\mathbb{R}}(2s_0) & \qquad \mathsf{f}_{\mathbb{R}}(2s_0) >_{\mathbb{R}_0,s_0} 5s_0
\end{array}$$

By definition of $>_{\mathbb{R}_0,s_0}$, these inequalities give rise to the following system of equations:

$$\mathsf{f}_{\mathbb{R}}(0) = 0 \qquad \mathsf{f}_{\mathbb{R}}(s_0) = s_0 \qquad \mathsf{f}_{\mathbb{R}}(2s_0) = 6s_0$$

After unraveling the definition of $\mathsf{f}_{\mathbb{R}}$ and substituting $z := as_0$, we get a system of linear equations in the unknowns z, b and c

$$c = 0 \qquad z + b = 1 \qquad 4z + 2b = 6$$

which has the unique solution $z = 2$, $b = -1$ and $c = 0$. Hence, $\mathsf{f}_{\mathbb{R}}$ must have the shape $\mathsf{f}_{\mathbb{R}}(x) = ax^2 - x = ax(x - \frac{1}{a})$ in every compatible polynomial interpretation over $\mathbb{R}$. However, this function is not a permissible interpretation for the function symbol f because it is not non-negative for all $x \in \mathbb{R}_0$. In particular, it is negative in the open interval $(0, \frac{1}{a})$; e.g., $\mathsf{f}_{\mathbb{R}}(\frac{1}{2a}) = -\frac{1}{4a}$. Hence, $\mathcal{S}$ is not compatible with any polynomial interpretation over $\mathbb{R}$. ∎

Remark 4.5. In this proof the interpretation of f is fixed to $\mathsf{f}_{\mathbb{R}}(x) = ax^2 - x$, which violates well-definedness in $\mathbb{R}_0$. However, this function is obviously well-defined in $\mathbb{R}_m$ for a properly chosen negative real number m. So, what happens if we take this $\mathbb{R}_m$ instead of $\mathbb{R}_0$ as carrier of a polynomial interpretation? To this end, we observe that $\mathsf{f}_{\mathbb{R}}(0) = 0$ and $\mathsf{f}_{\mathbb{R}}(\delta) = \delta(a\delta - 1) = \delta(as_0 - 1) = \delta$. Now let us consider some negative real number $x_0 \in \mathbb{R}_m$. Then $\mathsf{f}_{\mathbb{R}}(x_0) >_{\mathbb{R}} 0$ such that $\mathsf{f}_{\mathbb{R}}(\delta) - \mathsf{f}_{\mathbb{R}}(x_0) <_{\mathbb{R}} \delta$, which means that $\mathsf{f}_{\mathbb{R}}$ violates monotonicity with respect to the order $>_{\mathbb{R}_m,\delta}$.

The previous lemma, together with Theorem 3.2, yields the following corollary.

Corollary 4.6. *The TRS $\mathcal{S}$ is not polynomially terminating over $\mathbb{Q}$.*

Finally, combining the results presented in this section, we establish the main theorem of this paper.

Theorem 4.7. *There are TRSs that can be proved polynomially terminating over $\mathbb{N}$, but cannot be proved polynomially terminating over $\mathbb{R}$ or $\mathbb{Q}$.*

5. Conclusion and Future Work

In this paper, we investigated the relationship of polynomial interpretations with real, rational and integer coefficients with respect to termination proving power. In particular, we presented two new results, the first of which shows that polynomial interpretations with real coefficients subsume polynomial interpretations with rational coefficients, and the second of which shows that polynomial interpretations with real or rational coefficients do not properly subsume polynomial interpretations with integer coefficients, a result that comes somewhat unexpected. Together with the results of Lucas [Luc06], our results imply that polynomial interpretations with real or rational coefficients are incomparable to polynomial interpretations with integer coefficients with respect to termination proving power. Notwithstanding all these facts, the overall picture is not quite complete yet, there is still an open question: Are there TRSs that are polynomially terminating over $\mathbb{N}$ and $\mathbb{R}$, but not over $\mathbb{Q}$? Graphically, this question amounts to the inhabitation of the area depicted in red in Figure 2, which summarizes our results and the results of Lucas [Luc06].

We conclude this paper with two additional observations. First, we show that for polynomial interpretations over $\mathbb{R}$ it suffices to consider real *algebraic*[4] numbers as interpretation domain. Second, we present an alternative proof of Theorem 4.1, which shows the inhabitation of the area with the symbol $\mathbb{Q}$ in Figure 2.

Concerning the use of real algebraic numbers in polynomial interpretations, in [Luc07, Section 6] it is shown that it suffices to consider polynomials with real algebraic coefficients as interpretations of function symbols. Now the obvious question is whether it is also sufficient to consider only the (non-negative) real algebraic numbers $\mathbb{R}_{\mathsf{alg}}$ instead of the

[4] A real number is said to be algebraic if it is a root of a non-zero polynomial in one variable with rational coefficients.

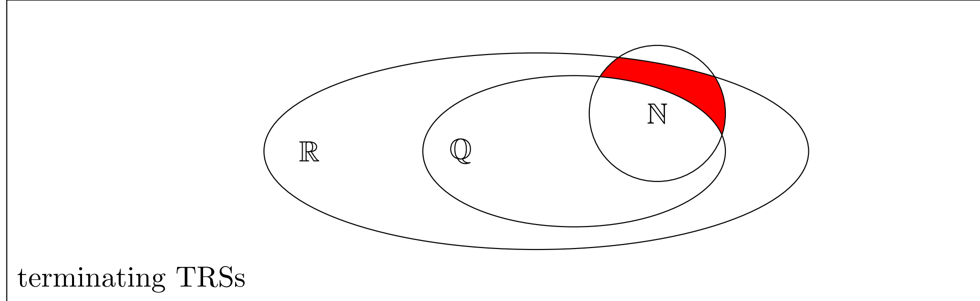


Figure 2: Comparison.

entire set $\mathbb{R}$ of real numbers as interpretation domain. We give an affirmative answer to this question by extending the result of [Luc07]. To this end, let us assume that $\mathcal{R}$ is a TRS that is polynomially terminating over $\mathbb{R}$. So, using the result of [Luc07], there exist a positive real number δ and a polynomial $f_{\mathbb{R}} \in \mathbb{R}_{\mathsf{alg}}[x_1, \ldots, x_n]$ for every n-ary function symbol $f \in \mathcal{F}$ such that:

(a) for all n-ary $f \in \mathcal{F}$, $f_{\mathbb{R}}(x_1, \ldots, x_n) \geq 0$ for all $x_1, \ldots, x_n \in \mathbb{R}_0$,
(b) for all $f \in \mathcal{F}$, $f_{\mathbb{R}}$ is strictly monotone with respect to $>_{\mathbb{R}_0,\delta}$ in all arguments,
(c) for every rewrite rule $l \to r \in \mathcal{R}$, $P_l >_{\mathbb{R}_0,\delta} P_r$ for all $x_1, \ldots, x_m \in \mathbb{R}_0$.

Treating δ as a variable (which we will later quantify existentially), we note that, similarly to the proof of Theorem 3.2, all three conditions can be phrased as quantified polynomial inequalities of the shape "$P(x_1, \ldots, x_k, \delta) \geq 0$ for all $x_1, \ldots, x_k \in \mathbb{R}_0$" for some polynomial P with real algebraic coefficients. Moreover, we note that there are finitely many of them if we assume $\mathcal{R}$ to be a finite TRS over a finite signature. Next we observe that any of these quantified inequalities can readily be expressed as a formula in the first order theory of ordered fields (where the atoms are polynomial (in)equalities, cf. [Bas06]) with δ as only free variable. Taking the conjunction of all these formulas and existentially quantifying δ and adding the conjunct $\delta > 0$, we obtain a sentence S in the first order theory of ordered fields, where all coefficients are real algebraic numbers. By assumption, this sentence holds in $\mathbb{R}$, and since both $\mathbb{R}$ and $\mathbb{R}_{\mathsf{alg}}$ are real closed fields with $\mathbb{R}_{\mathsf{alg}} \subset \mathbb{R}$ and all coefficients in S are from $\mathbb{R}_{\mathsf{alg}}$, we may apply the Tarski-Seidenberg transfer principle ([Bas06, Theorem 2.80]), from which we infer that S holds in $\mathbb{R}$ if and only if it holds in $\mathbb{R}_{\mathsf{alg}}$. Hence S also holds in $\mathbb{R}_{\mathsf{alg}}$ and therefore the TRS $\mathcal{R}$ is polynomially terminating over $\mathbb{R}_{\mathsf{alg}}$ (whose formal definition is the obvious specialization of Definition 2.5). This shows that polynomial termination over $\mathbb{R}$ implies polynomial termination over $\mathbb{R}_{\mathsf{alg}}$. As the reverse implication can be shown to hold by the same technique, we conclude that polynomial termination over $\mathbb{R}$ is equivalent to polynomial termination over $\mathbb{R}_{\mathsf{alg}}$.

Finally, we present our proof of Theorem 4.1, which is both shorter and simpler than the original proof in [Luc06, pp. 62–67]. Moreover, it shows that the strict inclusion holds even for *ground* TRSs.

Proof of Theorem 4.1. Consider the TRS $\mathcal{T}$ comprising the two rewrite rules

$$\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b}) \qquad\qquad \mathsf{g}(\mathsf{b}) \to \mathsf{g}(\mathsf{a})$$

We claim that $\mathcal{T}$ is polynomially terminating over $\mathbb{Q}$, but not over $\mathbb{N}$. We start with the latter. In every compatible polynomial interpretation over $\mathbb{N}$, we have $\mathsf{a}_{\mathbb{N}} > \mathsf{b}_{\mathbb{N}}$ or $\mathsf{a}_{\mathbb{N}} \leq \mathsf{b}_{\mathbb{N}}$.

Strict monotonicity of $\mathsf{f}_{\mathbb{N}}$ and $\mathsf{g}_{\mathbb{N}}$ yields $\mathsf{g}_{\mathbb{N}}(\mathsf{a}_{\mathbb{N}}) > \mathsf{g}_{\mathbb{N}}(\mathsf{b}_{\mathbb{N}})$ or $\mathsf{f}_{\mathbb{N}}(\mathsf{a}_{\mathbb{N}}) \leq \mathsf{f}_{\mathbb{N}}(\mathsf{b}_{\mathbb{N}})$. In both cases compatibility is violated. It remains to show that $\mathcal{T}$ is polynomially terminating over $\mathbb{Q}$. The following interpretation applies:

$$\delta := 1 \qquad \mathsf{a}_{\mathbb{Q}} := 0 \qquad \mathsf{b}_{\mathbb{Q}} := \tfrac{1}{2} \qquad \mathsf{g}_{\mathbb{Q}}(x) := 2x \qquad \mathsf{f}_{\mathbb{Q}}(x) := 6x^2 - 5x + 2$$

First, we show compatibility of this interpretation with the rules of $\mathcal{T}$. To this end, we observe that the inequalities

$$\mathsf{f}_{\mathbb{Q}}(\mathsf{a}_{\mathbb{Q}}) >_{\mathbb{Q}_0,\delta} \mathsf{f}_{\mathbb{Q}}(\mathsf{b}_{\mathbb{Q}}) \qquad\qquad \mathsf{g}_{\mathbb{Q}}(\mathsf{b}_{\mathbb{Q}}) >_{\mathbb{Q}_0,\delta} \mathsf{g}_{\mathbb{Q}}(\mathsf{a}_{\mathbb{Q}})$$

which simplify to $2 >_{\mathbb{Q}_0,1} 1$ and $1 >_{\mathbb{Q}_0,1} 0$, do indeed hold by definition of $>_{\mathbb{Q}_0,1}$. Next we show well-definedness (non-negativity) and monotonicity of $\mathsf{f}_{\mathbb{Q}}$ and $\mathsf{g}_{\mathbb{Q}}$.

For well-definedness we have to show $\mathsf{f}_{\mathbb{Q}}(x) \geq 0$ and $\mathsf{g}_{\mathbb{Q}}(x) \geq 0$ for all non-negative rational numbers x. While $\mathsf{g}_{\mathbb{Q}}$ obviously satisfies this condition, $\mathsf{f}_{\mathbb{Q}}$ requires further reasoning. To this end, it suffices to observe that $\mathsf{f}_{\mathbb{Q}}$ has a global minimum at $x_0 = \frac{5}{12}$, namely $\mathsf{f}_{\mathbb{Q}}(x_0) = \frac{23}{24}$, which is positive.

The strict monotonicity of $\mathsf{g}_{\mathbb{Q}}$ follows from Lemma 2.7. The function $\mathsf{f}_{\mathbb{Q}}$ is strictly monotone with respect to $>_{\mathbb{Q}_0,\delta}$ if and only if $\mathsf{f}_{\mathbb{Q}}(x + h) - \mathsf{f}_{\mathbb{Q}}(x) \geq \delta$ for all non-negative rational numbers x and $h \geq \delta$. Thus, we have to show that $h(6h - 5 + 12x) \geq 1$ for all non-negative rational numbers x and $h \geq 1$. As x is non-negative and occurs only with a positive sign, this is equivalent to showing that $h(6h - 5) \geq 1$ for all non-negative rational numbers $h \geq 1$, which is easy. Note that $\mathsf{f}_{\mathbb{Q}}$ is not strictly monotone with respect to the standard order $>_{\mathbb{Q}}$ on $\mathbb{Q}$. ∎

References

[Art00] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[Baa98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[Bas06] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Springer, 2nd edn., 2006.

[Con05] E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[Der79] N. Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979.

[Gie05] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)*, *Lecture Notes in Artificial Intelligence*, vol. 3452, pp. 301–331. 2005.

[Hir05] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

[Hof01] D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, *Lecture Notes in Computer Science*, vol. 2051, pp. 108–121. 2001.

[Kor03] K. Korovin and A. Voronkov. Orienting rewrite rules with the Knuth-Bendix order. *Information and Computation*, 183:165–186, 2003.

[Lan79] D. Lankford. On proving term rewrite systems are noetherian. Tech. Rep. MTP-3, Louisiana Technical University, Ruston, 1979.

[Lep01] I. Lepper. Derivation lengths and order types of Knuth-Bendix orders. *Theoretical Computer Science*, 269(1-2):433–450, 2001.

[Luc05] S. Lucas. Polynomials over the reals in proofs of termination: From theory to practice. *Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[Luc06] S. Lucas. On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 17(1):49–73, 2006.

[Luc07] S. Lucas. Practical use of polynomials over the reals in proofs of termination. In *Proc. 9th International Conference on Principles and Practice of Declarative Programming (PPDP 2007)*, pp. 39–50. Association of the Computing Machinery, 2007.

[Ter03] Terese. *Term Rewriting Systems*, *Cambridge Tracts in Theoretical Computer Science*, vol. 55. Cambridge University Press, 2003.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 259-276
http://rewriting.loria.fr/rta/

AUTOMATED TERMINATION ANALYSIS OF JAVA BYTECODE BY TERM REWRITING

CARSTEN OTTO AND MARC BROCKSCHMIDT AND CHRISTIAN VON ESSEN AND JÜRGEN GIESL

LuFG Informatik 2, RWTH Aachen University, Germany
E-mail address: giesl@informatik.rwth-aachen.de

Abstract. We present an automated approach to prove termination of Java Bytecode (JBC) programs by automatically transforming them to term rewrite systems (TRSs). In this way, the numerous techniques and tools developed for TRS termination can now be used for imperative object-oriented languages like Java, which can be compiled into JBC.

1. Introduction

Termination of TRSs and logic programs has been studied for decades. But as imperative programs dominate in practice, recently many results on termination of imperative programs were developed as well (e.g., [2, 3, 4, 5, 12]). Our goal is to re-use the wealth of techniques and tools from TRS termination when tackling imperative object-oriented programs. Similar TRS-based approaches have already proved successful for termination analysis of Prolog and Haskell [10, 17]. A first approach to prove termination of imperative programs by transforming them to TRSs was presented in [7]. However, [7] only analyzes a toy programming language without heap, whereas our goal is to analyze JBC programs.

JBC [14] is an assembly-like object-oriented language designed as intermediate format for the execution of Java [11] programs by a Java Virtual Machine (JVM). Moreover, JBC is a common compilation target for many other languages besides Java. While there exist several static analysis techniques for JBC, we are only aware of two other automated methods to analyze termination of JBC, implemented in the tools COSTA [1] and Julia [19]. They transform JBC into a constraint logic program by abstracting every object of a dynamic data type to an integer denoting its path-length (i.e., the maximal length of the path of pointers that can be obtained by following the fields of objects). For example, consider a data structure `IntList` with the field `value` for the first list element and the field `next` which points to the next list element. Now an object of type `IntList` representing the list $[0, 1, 2]$ would be abstracted to its length 3, but one would disregard the values of the list elements. While this fixed mapping from data objects to integers leads to a very efficient analysis, it also restricts the power of these methods. In contrast, in our approach

Key words and phrases: Java Bytecode, termination, term rewriting.

Supported by the DFG grant GI 274/5-2 and by the G.I.F. grant 966-116.6.

© C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.259

we represent data objects not by integers, but by *terms*. To this end, we introduce a function symbol for every class. So the `IntList` object above is represented by a term like $\mathsf{IntList}(0, \mathsf{IntList}(1, \mathsf{IntList}(2, \mathsf{null})))$, which keeps the whole information of the data object.

So compared to [1, 19] and to direct termination analysis of imperative programs, rewrite techniques[1] have the advantage that they are very powerful for algorithms on user-defined data structures, since they can automatically generate suitable well-founded orders comparing arbitrary forms of terms. Moreover, by using TRSs with built-in integers [8], rewrite techniques are also powerful for algorithms on pre-defined data types like integers.

Inspired by our approach for termination of Haskell [10], in this paper we present a method to translate JBC programs to TRSs. More precisely, in Sect. 2 we show how to automatically construct a *termination graph* representing all execution paths of the JBC program. Similar graphs are also used in program optimization techniques, e.g. [18]. While we perform considerably less abstraction than [1, 19], we also apply a suitable abstract interpretation [6] in order to obtain finite representations for all possible forms of the heap at a certain state. In contrast to *control flow graphs*, the nodes of the termination graph contain not just the current program position, but also detailed information on the values of the variables and on the content of the heap. Thus, the termination graph usually has several nodes which represent the same program position, but where the values of the variables and the heap are different. This is caused by different runs through the program code. The termination graph takes care of all aliasing, sharing, and cyclicity effects in the JBC program. This is needed in order to express these effects in a TRS afterwards. Then, a TRS is generated from the termination graph such that termination of the TRS implies termination of the original JBC program (Sect. 3). The resulting TRSs can be handled by existing TRS termination techniques and tools.

As described in Sect. 4, we implemented the transformation in our tool AProVE [9]. In the first *International Termination Competition* on automated termination analysis of JBC, AProVE achieved competitive results compared to Julia and COSTA. So this paper shows for the first time that rewriting techniques can indeed be successfully used for termination of imperative object-oriented languages like Java.

2. From JBC to Termination Graphs

To obtain a finite representation of all execution paths, we evaluate the JBC program symbolically, resulting in a *termination graph*. Afterwards, this graph is used to generate a TRS suitable for termination analysis. Sect. 2.1 introduces the abstract states used in termination graphs. Then Sect. 2.2 illustrates the construction of termination graphs for simple programs and Sect. 2.3 extends it to programs with complex forms of sharing.

[1] Of course, one could also use a transformation similar to ours where JBC is transformed to (constraint) logic programs, but where data objects are also represented by terms instead of integers. In principle, such an approach would be as powerful as ours, provided that one uses sufficiently powerful underlying techniques for automated termination analysis of logic programs. However, since some of the most powerful current termination analyzers for logic programs are based on term rewriting [15, 17], it seems more natural to transform JBC to term rewriting directly.

2.1. Representing States of the JVM

We define *abstract states* which represent *sets* of concrete JVM states, using a formalization which is especially suitable for a translation into TRSs (see e.g. [13] for related formalizations). Our approach is restricted to verified sequential JBC programs without recursion. To simplify the presentation in the paper, we only consider program runs involving a single method, and exclude floating point arithmetic, arrays, exceptions, and static class fields. However, our approach can easily be extended to such constructs and to arbitrary many non-recursive methods. For the latter, we represent the frames of the call stack individually and simply "inline" the code of invoked methods. Indeed, our implementation also handles programs with several methods including floats, arrays, exceptions, and static fields.

Definition 2.1. The set of abstract states is States = ProgPos × LocVar × OpStack × Heap.

The first component of a state corresponds to the program counter. We represent it by the next program instruction to be executed (e.g., by a JBC instruction like "`ifnull 8`").

The second component is an array of the local variables which have a defined value at the current program position, represented by a partial function LocVar = $\mathbb{N} \to$ References. Here, References are addresses in the heap. So in our representation, we do not store primitive values directly, but indirectly using references to the heap. This enables us to retain equality information for two otherwise unknown primitive values. Moreover, we require `null` $\in$ References to represent the `null` reference. To ease readability, in examples we usually denote local variables by names instead of numbers. Thus, "$\texttt{o}: o_1, \texttt{l}: o_2$" denotes an array where the 0-th local variable `o` references the address o_1 in the heap and the 1-st local variable `l` references the address o_2 in the heap. Of course, different local variables can point to the same address (e.g., in "$\texttt{o}:o_1, \texttt{l}:o_2, \texttt{c}:o_1$", `o` and `c` refer to the same object).

The third component is the operand stack that JBC instructions operate on. It will be filled with intermediate values such as operands of arithmetic operations when evaluating the bytecode. We represent it by a partial function OpStack = $\mathbb{N} \to$ References. The empty operand stack is denoted by "ε" and "i_1, i_2" denotes a stack with top element i_2.

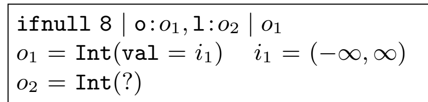


Figure 1: An abstract JVM state

To depict abstract states in examples, we write the first three components in the first line and separate them by "|". The fourth Heap component is written in the lines below, cf. Fig. 1. It describes the values of References. We represent the Heap by a partial function Heap : References $\to$ Integers $\cup$ Instances $\cup$ Unknown.

The values in Unknown = Classnames $\times \{?\}$ represent tree-shaped (and thus acyclic) objects for which we have no information except their type. Classnames contains the names of all classes and interfaces of the program. So for a class `Int`, "$o_2 = \texttt{Int}(?)$" means that the object at address o_2 is `null` or an instance of type `Int` (or a subtype of `Int`).

We represent integers as possibly unbounded intervals, i.e. Integers = $\{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. So $i_1 = (-\infty, \infty)$ means that any integer can be at the address i_1. Since current TRS termination tools cannot handle 32-bit `int`-numbers as in JBC, we treat `int` as the infinite set of all integers, i.e., we cannot handle problems related to overflows. Note that in JBC, `int` is also used for Boolean values.

To represent Instances (i.e., objects) of some class, we describe the values of their fields, i.e., Instances = Classnames $\times$(FieldIdentifiers $\to$ References). To prevent ambiguities, in general the FieldIdentifiers also contain the respective class names. So if the class `Int` has the field `val` of type `int`, then "$o_1 = \texttt{Int}(\texttt{val} = i_1)$" means that at the

```
00: aload_0       // load orig to opstack
01: ifnull 8      // jump to line 8 if top
                  // of opstack is null
04: aload_1       // load limit
05: ifnonnull 9   // jump if not null
08: return
09: aload_0       // load orig
10: astore_2      // store into copy
11: aload_0       // load orig
12: getfield val  // load field val
15: aload_1       // load limit
16: getfield val  // load field val
19: if_icmpge 35  // jump if
                  // orig.val >= limit.val
22: aload_2       // load copy
23: aload_2       // load copy
24: getfield val  // load field val
27: iconst_1      // load constant 1
28: iadd          // add copy.val and 1
29: putfield val  // store into copy.val
32: goto 11
35: return
```

(a) Java Bytecode

```
public class Int {
  // only wrap a primitive int
  private int val;

  // count up to the value
  // in "limit"
  public static void count(
      Int orig, Int limit) {

    if (orig == null
        || limit == null) {
      return;
    }

    // introduce sharing
    Int copy = orig;

    while (orig.val < limit.val) {
      copy.val++;
    }
  }
}
```

(b) Java Source Code

Figure 2: Example using aliasing and an integer counting upwards

address o_1, there is an instance of class `Int` and its field `val` references the address i_1 in the heap. Note that all sharing and aliasing must be explicitly represented in the abstract state. So since the state in Fig. 1 contains no sharing information for o_1 and o_2, o_1 and the references reachable from o_1 are disjoint from o_2 and from the references reachable from o_2.

2.2. Termination Graphs for Simple Programs

We now introduce the *termination graph* using a simple example. In Fig. 2(a) we present the analyzed JBC program and Fig. 2(b) shows the corresponding Java source code.

We create the termination graph using the states of a run of our abstract virtual machine as nodes, starting in a suitable general state. In our example, we want to know if *all* calls of the method `count` with two distinct arbitrary `Int` objects (or `null`) as arguments terminate. Here it is important to handle the aliasing of the variables `copy` and `orig`.

In Fig. 3, node A contains the start state. For the local variables `orig` and `limit` (abbreviated `o` and `l`), we only know their type and we know that they do not share any part of the heap. The first JBC instruction `aload_0` loads the value of the 0-th local variable (the argument `orig`) on the operand stack. The variable `orig` references some address o_1 in the heap, but we do not need concrete information about o_1 for this instruction. The resulting new state B is connected to A by an *evaluation edge*.

To evaluate the `ifnull` instruction, we need to know if the reference on top of the operand stack is `null`. This is not yet known for o_1. We *refine* the information and create successor nodes C and D for all possible cases (i.e., for o_1 == `null`, and for `Int` and all its non-abstract subclasses). In C, o_1 is `null`, and in D it is an instance of `Int` (`Int` has no proper subtypes). In D, the field values are new references in the heap. So instead of "o_1 = `Int`(?)", we now have "o_1 = `Int`(`val` = i_1)". Note that while "o_1 = `Int`(?)" in node B means that if o_1 is not `null`, then it has type `Int` or a subtype of it, "o_1 = `Int`(`val` = i_1)" in node D means that o_1's type is exactly `Int` and not a proper subtype. We have no information about the value at i_1. Therefore, i_1 gets the most general value for INTEGERS, i.e., $i_1 = (-\infty, \infty)$. C and D are connected to B by *refinement edges*.

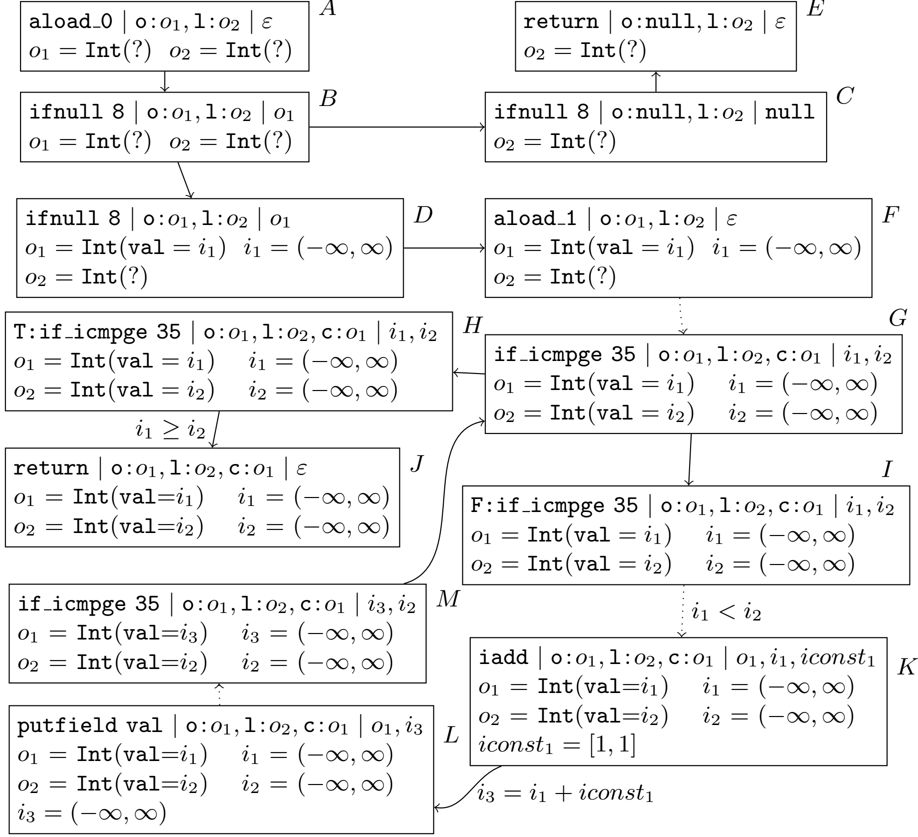


Figure 3: Termination graph for `count`

Now we can evaluate the instruction both for C and D, leading to E and F. Evaluation stops in E, while for F, the same procedure is repeated for the argument `limit`, leading to node G (among others) after several steps, indicated by a dotted arrow. Note the aliasing between `copy` and `orig`, since both reference the same object at the address o_1.

In G, we have already evaluated the two "`getfield val`" instructions and have pushed the two integer values on the operand stack. Now `if_icmpge` requires us to compare the unknown integers at i_1 and i_2. If we had to compare i_1 with a fixed number like 0, we could refine the information about i_1 and i_2 and create two successor nodes with $i_1 = (-\infty, -1]$ and $i_1 = [0, \infty)$. But "$i_1 \geq i_2$" is not expressible in our abstract states. Here, we split according to both possible values of the condition (depicted using the labels "`T`" and "`F`", respectively). This leads to the nodes H and I which are connected to G by *split edges*.

We can evaluate the condition in H to `true` and label the resulting evaluation edge to J by this condition. We will use these labels when constructing a TRS from the termination graph. J marks the program end and thus, it remains a leaf of the graph.

In I, we can evaluate the condition to `false` and label the next edge by the converse of the condition. After evaluating the next four instructions we reach node K. On the top positions of the operand stack, there are two integer variables (where the topmost variable has the value 1). The instruction `iadd` adds these two variables resulting in a new integer variable i_3. The relation between i_3, i_1, and $iconst_1$ is added as a label on the evaluation edge to the new node L. This label will again be used in the TRS construction.

From L on, we evaluate instructions until we again arrive at the instruction `if_icmpge` in node M. It turns out that M is an *instance* of the previous node G. Hence, we can connect M with G by an *instantiation edge*. The reason is that every concrete state which would be described by the abstract state M could also be described by the state G.

One has to expand termination graphs until all leaves correspond to program ends. Hence, our graph is now completed. By using appropriate generalization steps (which transform nodes into more general ones), one can always obtain a finite termination graph. To this end, one essentially executes the program symbolically until one reaches some position in the program for the second time. Then, a new state is created that is a generalization of both original states and one introduces instantiation edges from the two original states to the new generalized state. Of course, in our implementation we apply suitable heuristics to ensure that one only performs finitely many such generalization steps and to guarantee that the construction always terminates with a finite termination graph.

To define "*instance*" formally, we first define all positions π of references in a state s, where $s|_\pi$ denotes the reference at position π. A position π is a sequence starting with LV_n or OS_n for some $n \in \mathbb{N}$ (indicating the n-th reference in the local variable array or in the operand stack), followed by zero or more FieldIdentifiers.

Definition 2.2 (position, SPos)**.** Let $s = (pp, l, op, h) \in \text{States}$. Then $\text{SPos}(s)$ is the smallest set such that one of the following holds for all $\pi \in \text{SPos}(s)$:

- $\pi = \text{LV}_n$ for some $n \in \mathbb{N}$ where $l(n)$ is defined. Then $s|_\pi$ is $l(n)$.
- $\pi = \text{OS}_n$ for some $n \in \mathbb{N}$ where $op(n)$ is defined. Then $s|_\pi$ is $op(n)$.
- $\pi = \pi'\, v$ for some $v \in \text{FieldIdentifiers}$ and some $\pi' \in \text{SPos}(s)$ where $h(s|_{\pi'}) = (c, f) \in \text{Instances}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.

As an example, consider the state s depicted in node G of Fig. 3. Here we have three local variables and two elements on the operand stack. Thus, $\text{SPos}(s)$ contains $\text{LV}_0, \text{LV}_1, \text{LV}_2, \text{OS}_0, \text{OS}_1$, where $s|_{\text{LV}_0} = s|_{\text{LV}_2} = o_1$, $s|_{\text{LV}_1} = o_2$, $s|_{\text{OS}_0} = i_1$, and $s|_{\text{OS}_1} = i_2$. If h is the heap of that state, then $h(o_1) = (\texttt{Int}, f_1) \in \text{Instances}$, where $f_1(\texttt{val}) = i_1$. Hence, "$\text{LV}_0\, \texttt{val}$" is also a position in $\text{SPos}(s)$ and $s|_{\text{LV}_0\, \texttt{val}} = i_1$. The remaining elements of $\text{SPos}(s)$ are "$\text{LV}_2\, \texttt{val}$" and "$\text{LV}_1\, \texttt{val}$", where $s|_{\text{LV}_2\, \texttt{val}} = i_1$ and $s|_{\text{LV}_1\, \texttt{val}} = i_2$.

Intuitively, a state s' is an instance of a state s if they correspond to the same program position and whenever there is a reference $s'|_\pi$, then either the values represented by $s'|_\pi$ in the heap of s' are a subset of the values represented by $s|_\pi$ in the heap of s or else, π is no position in s. Moreover, shared parts of the heap in s' must also be shared in s. Note that since s and s' correspond to the same position in a *verified* JBC program, s and s' have the same number of local variables and their operand stacks have the same size.

Definition 2.3 (Instance)**.** We say that $s' = (pp', l', op', h')$ is an *instance* of state $s = (pp, l, op, h)$ (denoted $s' \sqsubseteq s$) iff $pp = pp'$, and for all $\pi, \pi' \in \text{SPos}(s')$:

(a) if $s'|_\pi = s'|_{\pi'}$ and $h'(s'|_\pi) \in \text{Instances} \cup \text{Unknown}$, then $\pi, \pi' \in \text{SPos}(s)$ and $s|_\pi = s|_{\pi'}$
(b) if $s'|_\pi \neq s'|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(s)$, then $s|_\pi \neq s|_{\pi'}$
(c) if $h'(s'|_\pi) \in \text{Integers}$ and $\pi \in \text{SPos}(s)$, then $h(s|_\pi) \in \text{Integers}$ and $h'(s'|_\pi) \subseteq h(s|_\pi)$
(d) if $s'|_\pi = \texttt{null}$ and $\pi \in \text{SPos}(s)$, then $s|_\pi = \texttt{null}$ or $h(s|_\pi) = (c, ?) \in \text{Unknown}$
(e) if $h'(s'|_\pi) = (c', ?)$ and $\pi \in \text{SPos}(s)$, then $h(s|_\pi) = (c, ?)$ where c' is c or a subtype of c
(f) if $h'(s'|_\pi) = (c', f') \in \text{Instances}$ and $\pi \in \text{SPos}(s)$, then $h(s|_\pi) = (c', f) \in \text{Instances}$ or $h(s|_\pi) = (c, ?)$, where c' must be c or a subtype of c.

The state s' in node M of Fig. 3 is an instance of the state s in node G. Clearly, they both refer to the same program position. It remains to examine the references reachable in s'. We have $\mathrm{SPos}(s') = \mathrm{SPos}(s) = \{\mathrm{LV}_0, \mathrm{LV}_1, \mathrm{LV}_2, \mathrm{OS}_0, \mathrm{OS}_1, \mathrm{LV}_0\,\mathtt{val}, \mathrm{LV}_1\,\mathtt{val}, \mathrm{LV}_2\,\mathtt{val}\}$. It is easy to check that the conditions of Def. 2.3 are satisfied for all these positions π. We illustrate this for $\pi = \mathrm{LV}_0\,\mathtt{val}$. Here, $s'|_\pi = i_3$ and if h' is the heap of s', then $h'(i_3) = (-\infty, \infty)$. Similarly, $s|_\pi = i_1$ and if h is the heap of s, then $h(i_1) = (-\infty, \infty)$. Here, s' and s are in fact equivalent, since M is an instance of G and G is an instance of M.

`if_icmpge35` | `o`:o_1, `l`:o_2, `c`:o_1 | i_1, i_2	
$o_1 = \mathtt{Int}(\mathtt{val}{=}i_1)$	$i_1 = [1, 1]$
$o_2 = \mathtt{Int}(\mathtt{val}{=}i_2)$	$i_2 = [10000, 10000]$

Figure 4: A concrete state

As remarked before, abstract states describe sets of *concrete states* like the one in Fig. 4, which is an instance of G and M. Here, the values for i_1 and i_2 are proper integers instead of intervals.

Definition 2.4 (Concrete state)**.** A state $s = (pp, l, op, h)$ is *concrete* if for all $\pi \in \mathrm{SPos}(s)$:

- $h(s|_\pi) \notin \mathrm{UNKNOWN}$ and
- if $h(s|_\pi) \in \mathrm{INTEGERS}$, then $h(s|_\pi)$ is just a singleton interval $[i, i]$ for some $i \in \mathbb{Z}$

A concrete state has no proper instances (i.e., if s is concrete and $s' \sqsubseteq s$, then $s \sqsubseteq s'$). Concrete states that are not a program end can always be evaluated and have exactly one (concrete) successor state. For Fig. 4, since i_1's value is not greater or equal than i_2's, the successor state corresponds to the instruction "`aload_2`", with the same local variables and empty operand stack. Such a sequence of concrete states, obtained by JBC evaluation, is called a *computation sequence*. Our construction of termination graphs ensures that

> if s is an abstract state in the termination graph and there is a concrete state $t \sqsubseteq s$ where t evaluates to the concrete state t', then the termination graph contains a path from s to a state s' with $t' \sqsubseteq s'$. (2.1)

To see why (2.1) holds, note that in the termination graph, s is first refined to a state $\overline{s}$ with $t \sqsubseteq \overline{s}$. So there is a path from s to $\overline{s}$, and in the state $\overline{s}$, all concrete information needed for an actual evaluation according to the JBC specification [14] is available. Note that "evaluation edges" in the termination graph are defined by exactly following the specification of JBC in [14]. Thus, there is an evaluation edge from $\overline{s}$ to s', where $t' \sqsubseteq s'$.

The computation sequence from Fig. 4 to its concrete successor corresponds to the path from node M or G to I's successor. Paths in the graph that correspond to computation sequences are called *computation paths*. Our goal is to show that all these paths are finite.

Definition 2.5 (Graph termination)**.** A finite or infinite path $s_1^1, \ldots, s_1^{n_1}, s_2^1, \ldots, s_2^{n_2}, \ldots$ through the termination graph is called a *computation path* iff there is a computation sequence $t_1, t_2, \ldots$ of concrete states where $t_i \sqsubseteq s_i^1$ for all i. A termination graph is called *terminating* iff it has no infinite computation path. Note that due to (2.1), if the termination graph is terminating, then the original JBC program is also terminating for all concrete states t where $t \sqsubseteq s$ for some abstract state s in the termination graph.

2.3. Termination Graphs for Complex Programs

Now we discuss sharing problems in complex programs with recursive data types. In Fig. 5, `flatten` takes a list of binary trees whose nodes are labeled by integers. It performs a depth-first run through all trees and returns the list of all numbers in these trees. It terminates because each loop iteration decreases the total number of all nodes in the trees of `list`, even though `list`'s length may increase. Note that `list` and `cur` share part of the heap.

```
public class Flatten {
  public static IntList flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;
    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
        IntList oldIntList = result;
        result = new IntList();
        result.value = tree.value;
        result.next = oldIntList;
        TreeList oldCur = cur;
        cur = new TreeList();
        cur.value = tree.left;
        cur.next = oldCur;
        oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

```
public class Tree {
  int value;
  Tree left;
  Tree right;
}

public class TreeList {
  Tree value;
  TreeList next;
}

public class IntList {
  int value;
  IntList next;
}
```

Figure 5: Example converting a list of binary trees to a list of integers

Consider the three states A, B, and C in Fig. 6. A is the state of our abstract JVM when it first reaches the loop condition "`cur != null`" (where `list`, `cur`, and `result` are abbreviated by `l`, `c`, and `r`). After one execution of the loop body, one obtains the state B if `tree` is `null` and C otherwise. Note that local variables declared in the loop body are no longer defined at the loop condition, and hence, they do not occur in A, B, or C.

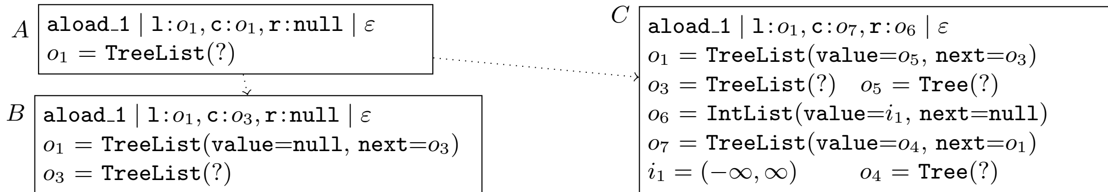


Figure 6: Three states of the termination graph of `flatten`

If one continued the evaluation like this, one would obtain an infinite tree, since one never reaches any state which is an instance of a previous state. (In particular, B and C are no instances of A.) Hence, to obtain *finite* graphs, one sometimes has to *generalize* states. Thus, we want to create a new general state S such that A, B, and C are instances of S. Note that in S, `l` and `c` cannot point to different references with UNKNOWN values, since then S would only represent states where `l` and `c` are tree-shaped and not sharing. However, `l` and `c` point to the *same* object in A, one can reach `c`: o_3 from `l`: o_1 in B (i.e., `l` *joins* `c`, since a field value of o_1 is o_3), and one can reach `l`: o_1 from `c`: o_7 in C. To express such sharing information in general states, we extend states by *annotations*.

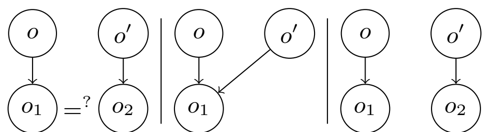


Figure 7: "$=^?$" annotation

In Fig. 7, the leftmost picture depicts a heap where an instance referenced by o has a field value o_1 and o' has a field value o_2. The annotation "$o_1 =^? o_2$" means that o_1 and o_2 could be equal. Here the value of at least one of o_1 and o_2 must be UNKNOWN. So both the second and the third shape

in Fig. 7 are instances of the first. In the second shape, o_1 and o_2 are equal and all occurrences of o_2 can be replaced by o_1 (or vice versa). In the third shape, o_1 and o_2 are not the same and thus, the annotation has been removed.

So the $=^?$ annotation covers both the equality of `l` and `c` in state A and their non-equality in states B and C. To represent states where `l` and `c` *may join*, we use the annotation "$\curlyvee$". We say that a reference o' is a *direct successor* of a reference o (denoted $o \to o'$) iff the object at address o has a field whose value is o'. As an example, consider state B in Fig. 6, where $o_1 \to o_3$ holds. Then the annotation "$o_1 \curlyvee o_2$" means that if o_1 is Unknown, then there could be an object o with $o_1 \to^+ o$ and $o_2 \to^* o$, i.e., o is a proper successor of o_1 and a (possibly non-proper) successor of o_2. Note that $\curlyvee$ is symmetric,[2] so $o_1 \curlyvee o_2$ also means that if o_2 is Unknown, then there could be an object o' with $o_1 \to^* o'$ and $o_2 \to^+ o'$. The shapes 2-4 in Fig. 8 visualize three possible instances of the state with annotation "$o_1 \curlyvee o_2$". Note that a state in which o_1 and o_2 do not share is also an instance.

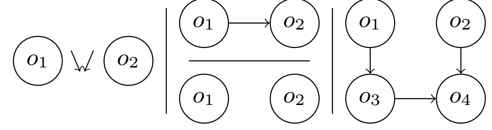


Figure 8: "$\curlyvee$" annotation

We can now create a state S (see Fig. 9) such that $A, B, C \sqsubseteq S$. The annotations state that `l` and `c` may be equal (as in A), that `l` may join `c` (as in B), or `c` may join `l` (as in C).

So to obtain a finite termination graph, after reaching A, we generalize it to a new node S connected by an instantiation edge. As seen in D, we introduce new forms of refinement edges to refine a state with the annotation "$o_1 =^? o_7$" into the two instances where $o_1 = o_7$ and where $o_1 \neq o_7$. For $o_1 = o_7$, we reach B' and C' which are like B and C but now `r` points to a list ending with o_6 instead of `null`. The nodes B' and C' are connected back to S with instantiation edges. For $o_1 \neq o_7$, due to `c != null`, we first refine the information about o_7, and obtain $o_7 = \texttt{TreeList}(\texttt{value} = o_8, \texttt{next} = o_9)$. Note that "$\curlyvee$" annotations have to be updated during refinements. If we have the annotation "$o_1 \curlyvee o_7$" and if one refines o_7 by introducing references like o_8, o_9 for its non-primitive fields, then we have to add corresponding annotations such as "$o_1 =^? o_9$" for all field references like o_9 whose types correspond to the type of o_1. Moreover, we add "$\curlyvee$" annotations for all non-primitive field references (i.e., "$o_1 \curlyvee o_8$" and "$o_1 \curlyvee o_9$"). If after this refinement neither o_1 nor o_7 were Unknown, we would delete the annotation $o_1 \curlyvee o_7$ since it has no effect anymore.

Now we use a refinement that corresponds to the case analysis whether `tree` is `null`. For `tree == null`, after one loop iteration we reach node E which is again an instance of S. Here, the local variable `tree` is no longer visible.

For `tree != null`, the graph shows nodes F and G. In F we need to evaluate a `putfield` instruction (corresponding to "`oldCur.value = tree.right`"), i.e., we have to put the object at address o_{11} to the field `value` of the object at address o_7. The effect of this operation can be seen in the box in state G, where the value of the object at o_7 was changed from o_8 to o_{11}. In G (which again corresponds to the loop condition), we removed the reference o_8 since it is no longer accessible from the local variables or the operand stack.

In contrast to other evaluation steps, such `putfield` instructions can give rise to additional annotations, since objects that already shared parts of the heap with o_7 now may also share parts of the heap with o_{11}. We say that a reference o *reaches* a reference o' iff there is a successor r of o (i.e., $o \to^* r$) such that $r = o'$ or $r =^? o'$ or $r \curlyvee o'$. So in our example, o_{11} reaches just o_{11} and o_1. Now if we write o_{11} to a field of o_7, then for all references o with $o \curlyvee o_7$, we have to add the annotation $o \curlyvee o'$ for all o' where o_{11} reaches o'. Hence,

[2]Since both "$=^?$" and "$\curlyvee$" are symmetric, we do not distinguish between "$o_1 =^? o_2$" and "$o_2 =^? o_1$" and we also do not distinguish between "$o_1 \curlyvee o_2$" and "$o_2 \curlyvee o_1$".

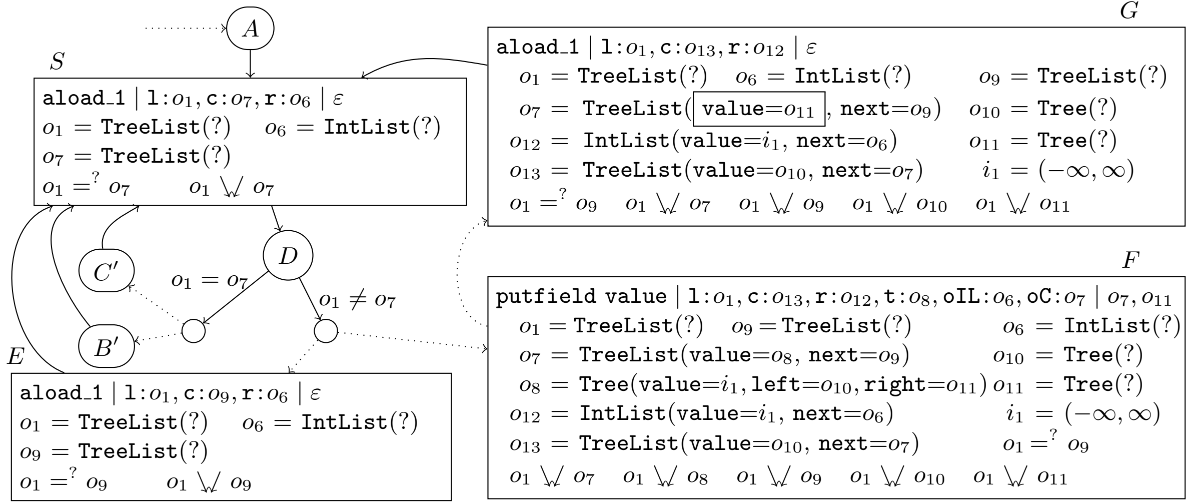


Figure 9: Termination graph for `flatten`

in our example, we would have to add $o_1 \searrow\!\swarrow o_{11}$ (which is already present in the state) and $o_1 \searrow\!\swarrow o_1$. However, the annotation $o_1 \searrow\!\swarrow o_1$ has no effect, since by "$o_1 = \texttt{TreeList}(?)$", we know that o_1 only represents tree-shaped objects. Therefore, we can immediately drop $o_1 \searrow\!\swarrow o_1$ from the state. Concrete non-tree-shaped objects can of course be represented easily (e.g., "$o = \texttt{TreeList}(\texttt{value} = o', \texttt{next} = o)$"). But to represent an *arbitrary* possibly non-tree-shaped object o, we use a special annotation (depicted "o!").[3]

Definition 2.6 (Instance & annotations)**.** We extend Def. 2.3 to $s, s' = (pp', l', op', h') \in$ States possibly containing annotations. Now $s' \sqsubseteq s$ holds iff for all $\pi, \pi' \in \text{SPos}(s')$, the following conditions are satisfied in addition to Def. 2.3 (b)-(f). Here, let τ resp. τ' be the maximal prefix of π resp. π' such that both $\tau, \tau' \in \text{SPos}(s)$.

(a) if $s'|_\pi = s'|_{\pi'}$ where $h'(s'|_\pi) \in \text{Instances} \cup \text{Unknown}$, and if $\pi, \pi' \in \text{SPos}(s)$,[4] then $s|_\pi = s|_{\pi'}$ or $s|_\pi =^? s|_{\pi'}$
(b) if $s'|_\pi =^? s'|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(s)$, then $s|_\pi =^? s|_{\pi'}$
(c) if $\big(s'|_\pi = s'|_{\pi'}$ where $h'(s'|_\pi) \in \text{Instances} \cup \text{Unknown}$, or $s'|_\pi =^? s'|_{\pi'}\big)$ and π or $\pi' \notin \text{SPos}(s)$ with $\pi \neq \pi'$, then $s|_\tau \searrow\!\swarrow s|_{\tau'}$
(d) if $s'|_\pi \searrow\!\swarrow s'|_{\pi'}$, then $s|_\tau \searrow\!\swarrow s|_{\tau'}$
(e) if $s'|_\pi!$ holds, then $s|_\tau!$
(f) if there exist (possibly empty) sequences $\rho \neq \rho'$ of FieldIdentifiers without common prefix, where $s'|_{\pi\rho} = s'|_{\pi\rho'}$, $h'(s'|_{\pi\rho}) \in \text{Instances} \cup \text{Unknown}$, and ($\pi\rho$ or $\pi\rho' \notin \text{SPos}(s)$ or $s|_{\pi\rho} =^? s|_{\pi\rho'}$), then $s|_\tau!$

3. From Termination Graphs to TRSs

Now we transform termination graphs into *integer term rewrite systems (ITRSs)*. These are TRSs where the Booleans $\mathbb{B}$, the integers $\mathbb{Z}$, and their built-in operations $\mathcal{A}rith\mathcal{O}p =$

[3] Such annotations can also result from `putfield` operations which write a reference o_2 to a field `f` of o_1. If o_1 already reached o_2 before through some field $\texttt{g} \neq \texttt{f}$, then we add "$o_1$!", since o_1 is no longer a tree. Even worse, if o_2 reached o_1 before, then `putfield` creates a cyclic object and we add "o_1!" and "o_2!".

[4] In contrast to Def. 2.3(a), here one may allow that π or $\pi' \notin \text{SPos}(s)$. This case is handled in (c).

$\{+,-,*,/,\%,\ll,\gg,\ggg,\hat{}\,,\&,|\}$ and $\mathcal{R}el\mathcal{O}p = \{>,\geq,<,\leq,==,\neq\}$ are pre-defined by an infinite set of variable-free rules $\mathcal{PD}$. For example, $\mathcal{PD}$ contains $1+2 \to 3$ and $5 < 4 \to \mathsf{false}$. As shown in [8], TRS termination techniques can easily be adapted to ITRSs as well.

Definition 3.1 (ITRS [8])**.** An *ITRS* is a finite conditional TRS with rules "$\ell \to r \mid b$". Here ℓ, r, b are terms, where $\ell \notin \mathbb{B} \cup \mathbb{Z}$ and ℓ contains no symbol from $\mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p$. However, b and r may contain extra variables not occurring in ℓ. We often omit the condition b if b is true. The *rewrite relation* $\hookrightarrow_{\mathcal{R}}$ of an ITRS $\mathcal{R}$ is the smallest relation where $t_1 \hookrightarrow_{\mathcal{R}} t_2$ iff there is a rule $\ell \to r \mid b$ from $\mathcal{R} \cup \mathcal{PD}$ such that $t_1|_p = \ell\sigma$, $b\sigma \hookrightarrow_{\mathcal{R}}^* \mathsf{true}$, and $t_2 = t_1[r\sigma]_p$. Here, $\ell\sigma$ must not have instances of left-hand sides of rules as proper subterms, and σ must be *normal* (i.e., $\sigma(y)$ is in normal form also for variables y occurring only in b or r). Thus, the rewrite relation $\hookrightarrow_{\mathcal{R}}$ corresponds to an *innermost* evaluation strategy.

So if $\mathcal{R}$ contains the rule "$\mathsf{f}(x) \to \mathsf{g}(x,y) \mid x > 2$", then $\mathsf{f}(1+2) \hookrightarrow_{\mathcal{R}} \mathsf{f}(3) \hookrightarrow_{\mathcal{R}} \mathsf{g}(3,27)$. Hence, extra variables in conditions or right-hand sides of rules stand for arbitrary values.

We first show how to transform a reference o in a state s into a term $\mathrm{tr}(s,o)$. References pointing to concrete integers like $iconst_1 = [1,1]$ in state K of Fig. 3 are transformed into the corresponding integer constant 1. The reference `null` is transformed into the constant null. References pointing to instances will be transformed by a refined transformation $\mathrm{ti}(s,o)$ in a more subtle way in order to take their types and the values of their fields into account. Finally, any other reference o is transformed into a variable (which we also call o). So $i_1 = (-\infty,\infty)$ in state K of Fig. 3 is transformed to the variable i_1.

Definition 3.2 (Transforming references)**.** Let $s = (pp, l, op, h) \in \textsc{States}$, $o \in \textsc{References}$.

$$\mathrm{tr}(s,o) = \begin{cases} i & \text{if } h(o) = [i,i], \text{ where } i \in \mathbb{Z} \\ \mathsf{null} & \text{if } o = \texttt{null} \\ \mathrm{ti}(s,o) & \text{if } h(o) \in \textsc{Instances} \\ o & \text{otherwise} \end{cases}$$

The main advantage of our approach becomes obvious when transforming instances (i.e., data objects) into terms. The reason is that data objects essentially *are* terms and we simply keep their structure when transforming them. So for any object, we use the name of its class as a function symbol. The arguments of the function symbol correspond to the fields of the class. As an example, consider o_{13} in state F of Fig. 9. This data object is transformed to the term $\mathsf{TreeList}(o_{10}, \mathsf{TreeList}(\mathsf{Tree}(i_1, o_{10}, o_{11}), o_9))$.

However, we also have to take the class hierarchy into account. Therefore, for any class c with n fields, let the corresponding function symbol now have arity $n+1$. The arguments $2, \ldots, n+1$ correspond to the values of the fields declared in class c. The first argument represents the part of the object that corresponds to subclasses of c. As an example, consider a class `A` with a field `a` of type `int` and a class `B` which extends `A` and has a field `b` of type `int`. If `x` is a data object of type `A` where `x.a` is `1`, then we now represent it by the term $\mathsf{A}(\mathsf{eoc}, 1)$. Here, the first argument of A is a constant eoc (for "end of class") which indicates that the type of `x` is really `A` and not a subtype of `A`. If `y` is a data object of type `B` where `y.a` is `2` and `y.b` is `3`, then we represent it by the term $\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2)$. So the class hierarchy is represented by nesting the function symbols corresponding to the respective classes.

More precisely, since every class extends `java.lang.Object` (which has no fields), each such term now has the form $\mathsf{java.lang.Object}(\ldots)$. Hence, if we abbreviate the function symbol $\mathsf{java.lang.Object}$ by jlO, then for the `TreeList` object above, now the corresponding term is $\mathsf{jlO}(\mathsf{TreeList}(\mathsf{eoc}, o_{10}, \mathsf{jlO}(\mathsf{TreeList}(\mathsf{eoc}, \mathsf{jlO}(\mathsf{Tree}(\mathsf{eoc}, i_1, o_{10}, o_{11}), o_9)))))$.

Of course, we can only transform tree-shaped objects to terms. If $\pi \in \text{SPos}(s)$ and if there is a non-empty sequence ρ of FieldIdentifiers such that $s|_\pi = s|_{\pi\rho}$, then $s|_\pi$ is called *cyclic* in s. If $s|_\pi$ is cyclic or marked by "!", then $s|_\pi$ is called *special*. Every special reference o is transformed into a variable o in order to represent an "arbitrary unknown" object. To define the transformation $\text{ti}(s, o)$ formally, we use an auxiliary transformation $\overline{\text{ti}}(s, o, c)$ which only considers the part of the class hierarchy starting with c.

Definition 3.3 (Transforming instances)**.** We start the construction at the root of the class hierarchy (i.e., with `java.lang.Object`) and define $\text{ti}(s, o) = \overline{\text{ti}}(s, o, \texttt{java.lang.Object})$.

Let $s = (pp, l, op, h) \in \text{States}$ and let $h(o) = (c_o, f) \in \text{Instances}$. Let $(c_1 = \texttt{java.lang.Object}, c_2, \ldots, c_n = c_o)$ be ordered according to the class hierarchy, i.e., c_i is the direct superclass of c_{i+1}. We define the term $\overline{\text{ti}}(s, o, c_i)$ as follows:

$$\overline{\text{ti}}(s, o, c_i) = \begin{cases} o & \text{if } o \text{ is special} \\ c_o(\mathsf{eoc}, \text{tr}(s, v_1), \ldots, \text{tr}(s, v_m)) & \text{if } c_i = c_o, \text{fv}(c_o, f) = v_1, \ldots, v_m \\ c_i(\overline{\text{ti}}(s, o, c_{i+1}), \text{tr}(s, v_1), \ldots, \text{tr}(s, v_m)) & \text{if } c_i \neq c_o, \text{fv}(c_i, f) = v_1, \ldots, v_m \end{cases}$$

For $c \in \text{Classnames}$, let `f1`, ..., `fm` be the fields declared in c in some fixed order. Then for $f : \text{FieldIdentifiers} \to \text{References}$, $\text{fv}(c, f)$ gives the values $f(\texttt{f1}), \ldots, f(\texttt{fm})$.

So for class `A` and `B` above, if $h(o) = (\texttt{B}, f)$ where $f(\texttt{a}) = [2, 2]$ and $f(\texttt{b}) = [3, 3]$, then $\text{fv}(\texttt{A}, f) = [2, 2]$, $\text{fv}(\texttt{B}, f) = [3, 3]$ and thus, $\overline{\text{ti}}(s, o, \texttt{java.lang.Object}) = \mathsf{jlO}(\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2))$.

To transform a whole state, we create the tuple of the terms that correspond to the references in the local variables and the operand stack. For example, state J in Fig. 3 is transformed to the tuple of the terms $\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$, $\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2))$, and $\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))$.

Definition 3.4 (Transforming states)**.** Let $s = (pp, l, op, h) \in \text{States}$, let $lv_0, \ldots, lv_n$ and $os_0, \ldots, os_m$ be the references in l and op, respectively (i.e., $h(\text{LV}_i) = lv_i$ and $h(\text{OS}_i) = os_i$). We define the following mapping: $\text{ts}(s) = (\text{tr}(s, lv_0), \ldots, \text{tr}(s, lv_n), \text{tr}(s, os_0), \ldots, \text{tr}(s, os_m))$.

There is a connection between the instance relation on states and the matching relation on the corresponding terms. If s' is an instance of state s, then the terms in the transformation of s match the terms in the transformation of s'. Hence, if one generates rules matching the term representation of s, then these rules also match the term representation of s'.

Lemma 3.5. *Let $s' \sqsubseteq s$. Then there exists a substitution σ such that* $\text{ts}(s)\sigma = \text{ts}(s')$.[5]

Now we show how to build an ITRS from a termination graph such that termination of the ITRS implies termination of the graph, i.e., that there is no infinite computation path $s_1^1, \ldots, s_1^{n_1}, s_2^1, \ldots, s_2^{n_2}, \ldots$. In other words, there should be no infinite computation sequence $t_1, t_2, \ldots$ of concrete states where $t_i \sqsubseteq s_i^1$ for all i.

For any abstract state s of the graph, we introduce a new function symbol f_s. The arity of f_s is the number of components in the tuple $\text{ts}(s)$. Our goal is to generate an ITRS $\mathcal{R}$ such that $\mathsf{f}_{s_i^1}(\text{ts}(t_i)) \hookrightarrow_{\mathcal{R}}^+ \mathsf{f}_{s_{i+1}^1}(\text{ts}(t_{i+1}))$ for all i. In other words, every computation path in the graph must be transformable into a rewrite sequence. Then each infinite computation path corresponds to an infinite rewrite sequence with $\mathcal{R}$.

To this end, we transform each edge in the termination graph into a rewrite rule. Let s, s' be two states connected by an edge e. If e is a split edge or an evaluation edge, then the corresponding rule should rewrite any instance of s to the corresponding instance of s'.

[5]For all proofs, we refer to [16].

Hence, we generate the rule $\mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s'))$. For example, the edge from D to F in Fig. 3 results in the rule $\mathsf{f}_D(\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), o_2, \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))) \to \mathsf{f}_F(\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), o_2)$. If the evaluation involves checking some integer condition, we create a corresponding conditional rule. For example, the edge from H to J in Fig. 3 yields the rule

$$\begin{array}{ll}\mathsf{f}_H(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)), \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), i_1, i_2) \to & \\ \mathsf{f}_J(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)), \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)), \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)) & \mid i_1 \geq i_2\end{array}$$

The only evaluation edges which do not result in the rule $\mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s'))$ are evaluations of `putfield` instructions. If `putfield` writes to the field of an object at reference o, then this could modify all objects at references o' with $o \searrow\!\!\swarrow o'$.[6] Therefore, in the right-hand side of the rule corresponding to `putfield`, we do not transform the reference o' to the variable o', but to a fresh variable $\overline{o'}$. As an example consider state F in Fig. 9, where we write to a field of o_7, and we have the annotation $o_1 \searrow\!\!\swarrow o_7$. In the resulting rule, we therefore have the variable o_1 on the left-hand side, but a fresh variable $\overline{o_1}$ on the right-hand side. The terms corresponding to o_7 on the left- and right-hand side of the resulting rule describe the update of its field precisely (i.e., $\mathsf{jlO}(\mathsf{Tree}(\mathsf{eoc}, i_1, o_{10}, o_{11}))$ is replaced by o_{11}).

Now let e be an instance edge from s to s'. Here we keep the information that we already have for the specialized state s (i.e., we keep $\text{ts}(s)$) and continue rewriting with the rules we already created for s'. So instead of $\mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s'))$ we generate $\mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s))$.

Finally, let e be a refinement edge from s to s'. So some abstract information in s is refined to more concrete information in s' (e.g., by refining $(c, ?)$ to `null`). These edges represent a case analysis and hence, some instances of s are also instances of s', but others are no instances of s'. Note that by Lemma 3.5, if a state t is an instance of s', then the term representation of s' matches t's term representation. Hence, we can use pattern matching to perform the necessary case analysis. So instead of the rule $f_s(\text{ts}(s)) \to f_{s'}(\text{ts}(s'))$, we create a rule whose left-hand side only matches instances of s', i.e., $f_s(\text{ts}(s')) \to f_{s'}(\text{ts}(s'))$. Consider for example the edge from B to C in Fig. 3. Any concrete state whose evaluation corresponds to this edge must have `null` at positions LV_0 and OS_0. Thus, we create the rule $\mathsf{f}_B(\mathsf{null}, o_2, \mathsf{null}) \to \mathsf{f}_C(\mathsf{null}, o_2, \mathsf{null})$ which is only applicable to such states.

Recall that possibly cyclic data objects are translated to variables in Def. 3.3. Although variables are only instantiated by finite (non-cyclic) terms in term rewriting, our approach remains sound because states with possibly cyclic objects result in rules with extra variables on right-hand sides. For example, consider a simple list traversal algorithm. Here, we would have a state s where the local variable points to a reference o_1 with $o_1 = \texttt{Intlist}(\texttt{value} = i_1, \texttt{next} = o_2)$ and in the successor state s', the local variable would point to o_2. Then, after refinement to $o_2 = \texttt{IntList}(\texttt{value} = i_2, \texttt{next} = o_3)$, there would be an instantiation edge back to s. For acyclic lists, this results in the rules $\mathsf{f}_s(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc}, i_1, o_2))) \to \mathsf{f}_{s'}(o_2)$, $\mathsf{f}_{s'}(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc}, i_2, o_3))) \to \mathsf{f}_{s''}(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc}, i_2, o_3)))$ and $\mathsf{f}_{s''}(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc}, i_2, o_3))) \to \mathsf{f}_s(\mathsf{jlO}(\mathsf{IntList}(\mathsf{eoc}, i_2, o_3)))$ whose termination is easy to show. But if we had the annotations $o_1!$ and $o_2!$ in s, $o_2!$ in s', and $o_2!$ and $o_3!$ in s'', then we would obtain the rules $\mathsf{f}_s(o_1) \to \mathsf{f}_{s'}(o_2)$, $\mathsf{f}_{s'}(o_2) \to \mathsf{f}_{s''}(o_2)$ and $\mathsf{f}_{s''}(o_2) \to \mathsf{f}_s(o_2)$. So in the first rule, o_2 would be an extra variable representing an arbitrary list, and the resulting rules would not be terminating.

Definition 3.6 (Rewrite rules from termination graphs)**.** Let there be an edge e from the state $s = (pp, l, op, h)$ to the state s' in a termination graph. Then we generate $rule(e)$:

- if e is an instance edge, then $rule(e) = \mathsf{f}_s(\text{ts}(s)) \to \mathsf{f}_{s'}(\text{ts}(s))$

[6] Note that this is only possible if o' is UNKNOWN.

• if e is a refinement edge, then $rule(e) = \mathsf{f}_s(\mathrm{ts}(s')) \to \mathsf{f}_{s'}(\mathrm{ts}(s'))$
• if e is an evaluation or split edge, we perform the following case analysis:
  – if e is labeled by a statement of the form $o_1 = o_2 \oplus o_3$ where $\oplus \in \mathcal{A}rith\mathcal{O}p$, then $rule(e) = \mathsf{f}_s(\mathrm{ts}(s)) \to \mathsf{f}_{s'}(\mathrm{ts}(s'))\sigma$, where σ substitutes o_1 by $\mathrm{tr}(s, o_2) \oplus \mathrm{tr}(s, o_3)$
  – if e is labeled by a condition $o_1 \oplus o_2$ where $\oplus \in \mathcal{R}el\mathcal{O}p$, then $rule(e) = \mathsf{f}_s(\mathrm{ts}(s)) \to \mathsf{f}_{s'}(\mathrm{ts}(s')) \mid \mathrm{tr}(s, o_1) \oplus \mathrm{tr}(s, o_2)$
  – if pp is the instruction `putfield` writing to a field of reference o, then $rule(e) = \mathsf{f}_s(\mathrm{ts}(s)) \to \mathsf{f}_{s'}(\mathrm{ts}_o(s'))$, where $\mathrm{ts}_o(s')$ is defined like $\mathrm{ts}(s')$, but each reference o' with $o \searrow\!\!\swarrow o'$ is transformed into a new fresh variable.
  – for all other instructions, $rule(e) = \mathsf{f}_s(\mathrm{ts}(s)) \to \mathsf{f}_{s'}(\mathrm{ts}(s'))$

Our main theorem states that every computation path of the termination graph can be simulated by a rewrite sequence using the corresponding ITRS. Of course, the converse does not hold, i.e., our approach cannot be used to prove non-termination of JBC programs.

Theorem 3.7 (Proving termination of JBC by ITRSs). *If the ITRS corresponding to a termination graph is terminating, then the termination graph is terminating as well. Hence, then the original JBC program is also terminating for all concrete states t where $t \sqsubseteq s$ for some abstract state s in the termination graph.*

The resulting ITRSs are usually large, since they contain one rule for each edge of the termination graph. But since our ITRSs have a special form where the roots of all left- and right-hand sides are defined, where defined symbols do not occur below the roots, and where we only consider rewriting with normal substitutions, one can simplify the ITRSs substantially by *merging* their rules: Let $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) be those rules in $\mathcal{R}$ where the root of the right- (resp. left-)hand side is f. Then one can replace the rules $\mathcal{R}_1 \cup \mathcal{R}_2$ by the rules "$\ell\sigma \to r'\sigma \mid b\sigma \;\&\&\; b'\sigma$" for all $\ell \to r \mid b \in \mathcal{R}_1$ and all $\ell' \to r' \mid b' \in \mathcal{R}_2$, where $\sigma = \mathrm{mgu}(r, \ell')$. Of course, we also have to add rules for the Boolean conjunction "&&". Clearly, this process does not modify the termination behavior of $\mathcal{R}$. Moreover, it suffices to create rules only for those edges that occur in cycles of the termination graph.

With this simplification, we automatically obtain the following 1-rule ITRS for the `count` program. It increases the value i_1 of f_G's first and third argument (corresponding to the `value`-field of `orig` and `copy`) as long as $i_1 < i_2$ (where i_2 is the `value`-field of `limit`).

$$\begin{array}{l} \mathsf{f}_G(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\quad \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)),\quad \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\quad i_1,\quad i_2) \;\to \\ \mathsf{f}_G(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1+1)),\quad \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_2)),\quad \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1+1)),\quad i_1+1,\quad i_2) \;\mid\; i_1 < i_2 \end{array}$$

For the `flatten` program we automatically obtain the following ITRS. To ease readability, we replaced every subterm "$\mathsf{jlO}(t)$" by just t, and we replaced "$\mathsf{TreeList}(\mathsf{eoc}, v, n)$" by "$\mathsf{TL}(v, n)$", "$\mathsf{Tree}(\mathsf{eoc}, v, l, r)$" by "$\mathsf{T}(v, l, r)$", and "$\mathsf{IntList}(\mathsf{eoc}, v, n)$" by "$\mathsf{IL}(v, n)$".

$$\mathsf{f}_S(\mathsf{TL}(\mathsf{null}, o_9), \mathsf{TL}(\mathsf{null}, o_9), o_6) \;\to\; \mathsf{f}_S(\mathsf{TL}(\mathsf{null}, o_9), o_9, o_6) \tag{3.1}$$

$$\mathsf{f}_S(\mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9), \mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9), o_6) \;\to\; \mathsf{f}_S(\mathsf{TL}(o_{11}, o_9), \mathsf{TL}(o_{10}, \mathsf{TL}(o_{11}, o_9)), \mathsf{IL}(i_1, o_6)) \tag{3.2}$$

$$\mathsf{f}_S(o_1, \mathsf{TL}(\mathsf{null}, o_9), o_6) \;\to\; \mathsf{f}_S(o_1, o_9, o_6) \tag{3.3}$$

$$\mathsf{f}_S(o_1, \mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9), o_6) \;\to\; \mathsf{f}_S(\overline{o_1}, \mathsf{TL}(o_{10}, \mathsf{TL}(o_{11}, o_9)), \mathsf{IL}(i_1, o_6)) \tag{3.4}$$

Rules (3.1) and (3.3) correspond to the cycles from S over B' and over E. Their difference is whether `l` and `c` point to the same object in S (i.e., whether the first two arguments of f_S in the left-hand side are identical). But both handle the case where the first tree in the list `c` (i.e., in f_S's second argument) is null. Then this null-tree is simply removed from the list and the result `r` (i.e., the third argument of f_s) does not change. The rules (3.2) and

(3.4) correspond to the cycles from S over C' and over G. Here, the list `c` has the form $\mathsf{TL}(\mathsf{T}(i_1, o_{10}, o_{11}), o_9)$. Hence, the value i_1 of the first tree in the list is stored in the result list (which is modified from o_6 to $\mathsf{IL}(i_1, o_6)$) and the list `c` is modified to $\mathsf{TL}(o_{10}, \mathsf{TL}(o_{11}, o_9))$. So the length of the list increases, but the number of nodes in the list decreases.

These examples illustrate that the ITRSs resulting from our automatic transformation of JBC are often very readable and constitute a natural representation of the original algorithm as a rewrite system. Not surprisingly, existing TRS techniques can easily prove termination of the resulting rules. For example, termination of the above ITRS for `flatten` is easily proved using a straightforward polynomial interpretation and dependency pairs. In contrast, abstraction-based tools like Julia and COSTA fail on examples like `flatten`. In fact, Julia and COSTA also fail on the `count` example from Sect. 2.2.

4. Experiments and Conclusion

We introduced an approach to prove termination of JBC programs automatically by first transforming them to termination graphs. Then an integer TRS is generated from the termination graph and existing TRS tools can be used to show its termination.

We implemented our approach in the termination prover AProVE [9] and evaluated it on the 106 non-recursive JBC examples from the *termination problem data base (TPDB)* used in the *International Termination Competition*.[7] In our experiments, we removed one controversial example ("`overflow`") from the TPDB whose termination depends on the treatment of integer overflows and we added the two examples `count` and `flatten` from this paper. Of these 106 examples, 10 are known to be non-terminating. See `http://aprove.informatik.rwth-aachen.de/eval/JBC` for the origins of the individual examples. As in the competition, we ran AProVE and the tools Julia [19] and COSTA [1] with a time limit of 60 seconds on each example. "Success" gives the number of examples where termination was proved, "Failure" means that the proof failed in less than 60 seconds, "Timeout" gives the number of examples where the tool took longer than 60 seconds, and "Runtime" is the average time (in s) needed per example. Note that for those examples from this collection where AProVE resulted in a timeout, the tool would also fail when using a longer timeout.

	all 106 non-recursive examples			
	Success	Failure	Timeout	Runtime
AProVE	89	5	12	14.3
Julia	74	32	0	2.6
COSTA	60	46	0	3.4

Our experiments show that for the problems in the current example collection, our rewriting-based approach in AProVE currently yields the most precise results. The main reason is that we do not use a fixed abstraction from data objects to integers, but represent objects as terms. On the other hand, this also explains the larger runtimes of AProVE compared to Julia and COSTA. Still, our approach is efficient enough to solve most examples in reasonable time. Our method benefits substantially from the representation of objects as terms, since afterwards arbitrary TRS termination techniques can be used to prove termination of the algorithms. Of course, while the examples in the TPDB are challenging, they are still quite small. Future work will be concerned with the application and adaption of our approach in order to use it also for large examples and Java libraries.

[7]See `http://www.termination-portal.org/wiki/Termination_Competition`.

In the current paper, we restricted ourselves to JBC programs without recursion, whereas the approaches of Julia and COSTA also work on recursive programs. Of course, an extension of our method to recursive programs is another main point for future work. Our experiments also confirm the results at the *International Termination Competition* in December 2009, where the first competition on termination of JBC programs took place. Here, the three tools above were run on a random selection of the examples from the TPDB with similar results. To experiment with our implementation via a web interface and for details about the above experiments, we refer to `http://aprove.informatik.rwth-aachen.de/eval/JBC`.

Acknowledgement

We are indebted to the Julia- and the COSTA-team for their help with the experiments. We thank the anonymous reviewers for their valuable comments.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java Bytecode. In *Proc. FMOODS '08*, LNCS 5051, pages 2–18, 2008.
[2] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*, LNCS 4144, pages 386–400, 2006.
[3] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of polynomial programs. In *Proc. VMCAI '05*, LNCS 3385, pages 113–129, 2005.
[4] M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV '02*, LNCS 2404, pages 442–454, 2002.
[5] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*, pages 415–426. ACM Press, 2006.
[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL '77*, pages 238–252. ACM Press, 1977.
[7] S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *Proc. CADE '09*, LNAI 5663, pages 277–293, 2009.
[8] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA '09*, LNCS 5595, pages 32–47, 2009.
[9] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. IJCAR '06*, LNAI 4130, pages 281–286, 2006.
[10] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *RTA '06*, LNCS 4098, pp. 297–312, 2006.
[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2005.
[12] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Proc. POPL '09*, pages 127–139. ACM Press, 2009.
[13] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
[14] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.
[15] M. T. Nguyen, D. De Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming*, 2010. To appear. Available from `http://arxiv.org/pdf/0912.4360`.
[16] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. Technical Report AIB-2010-08, RWTH Aachen, 2010. `http://aib.informatik.rwth-aachen.de`.
[17] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1), Article 2, 2009.

[18] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. ILPS '95*, pages 465–479. MIT Press, 1995.

[19] F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), Article 8, 2010.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 277-294
http://rewriting.loria.fr/rta/

DECLARATIVE DEBUGGING OF MISSING ANSWERS FOR MAUDE SPECIFICATIONS

ADRIÁN RIESCO [1] AND ALBERTO VERDEJO [1] AND NARCISO MARTÍ-OLIET [1]

[1] Facultad de Informática, Universidad Complutense de Madrid, Spain
E-mail address: `ariesco@fdi.ucm.es,{alberto,narciso}@sip.ucm.es`

Abstract. Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error by building a tree representing this computation and guiding the user through it to find the error. Membership equational logic (*MEL*) is an equational logic that in addition to equations allows to state of membership axioms characterizing the elements of a sort. Rewriting logic is a logic of change that extends *MEL* by adding rewrite rules, that correspond to transitions between states and can be nondeterministic. In this paper we propose a calculus to infer normal forms and least sorts with the equational part, and sets of reachable terms through rules. We use an abbreviation of the proof trees computed with this calculus to build appropriate debugging trees for missing answers (results that are erroneous because they are incomplete), whose adequacy for debugging is proved. Using these trees we have implemented a declarative debugger for Maude, a high-performance system based on rewriting logic, whose use is illustrated with an example.

1. Introduction

Declarative debugging [20], also known as declarative diagnosis or algorithmic debugging, is a debugging technique that abstracts the execution details, which may be difficult to follow in declarative languages, and focus on the results. We can distinguish between two different kinds of declarative debugging: debugging of *wrong answers*, that is applied when a *wrong* result is obtained from an initial value and has been widely employed in the logic [12, 22], functional [14, 15], multi-paradigm [3, 9], and object-oriented [4] programming languages; and debugging of *missing answers* [5, 1], applied when a result is *incomplete*, which has been less studied because the calculus involved is more complex than in the case of wrong answers. Declarative debugging starts from an incorrect computation, the error symptom, and locates the code (or the absence of code) responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [13], where each node stands for a computation step and must follow from the results of its child nodes by some logical inference. This tree is traversed by asking questions to an external oracle (generally the user)

1998 ACM Subject Classification: D.2.5 [Software Engineering]: Testing and Debugging – Debugging aids, D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications – Nondeterministic languages.
Key words and phrases: Declarative debugging, Maude, Missing answers, Rewriting Logic.
Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

© A. Riesco, A. Verdejo, and N. Martí-Oliet
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.277

until a *buggy node*—a node containing an erroneous result, but whose children are all correct—is found. Hence, we distinguish two phases in this scheme: the debugging tree *generation* and its *navigation* following some suitable strategy [21].

In this paper we present a declarative debugger of missing answers for *Maude specifications*. Maude [6] is a high-level language and high-performance system supporting both equational and rewriting logic computation. The Maude system supports several approaches for debugging: tracing, term coloring, and using an internal debugger [6, Chap. 22]. However, these tools have the disadvantages that they are supposed to be used only when a wrong result *is found*; and both the trace and the Maude debugger (that is based on the trace) show the statements applied in the order in which they are executed and thus the user can lose the general view of the *proof* of the incorrect computation that produced the wrong result.

Declarative debugging of wrong answers in Maude specifications was already studied in [17], where we presented how to debug wrong results due to errors in the statements of the specification. In [18] we extended the concept of missing answers, usually attached to incomplete sets of results, to deal with erroneous normal forms and least sorts in equational theories. However, in a nondeterministic context such as that of Maude modules other problems can arise. We show in this paper how to debug missing answers in rewriting specifications, that is, expected results that the specification is not able to compute. This kind of problems appears in Maude when using its breadth-first search, that finds all the reachable terms from an initial one given a pattern, a condition, and a bound in the number of steps. To debug this kind of errors we have extended our calculus to deduce sets of reachable terms given an initial term, a bound in the number of rewrites, and a condition to be fulfilled. Unlike other proposals like [5], our debugging framework combines the treatment of wrong and missing answers and, moreover, is able to detect missing answers due to both missing rules and wrong statements. The state of the art can be found in [21], where different algorithmic debuggers are compared and that will include our debugger in its next version. Roughly speaking, our debugger has the pros of building different kinds of debugging trees (one-step and many-steps) and applying the missing answers technique to debug normal forms and least sorts[1] (the different trees are a novelty in the declarative debugging world), and only it and DDT [3] implement the Hirunkitti's divide and query navigation strategy, provide a graphical interface, and debug missing answers; as cons, we do not provide answers like "maybe yes," "maybe not," and "inadmissible," and do not perform tree compression. However, these features have recently been introduced in specific debuggers, and we expect to implement them in our debugger soon. Finally, some of the features shared by most of the debuggers are: the trees are abbreviated in order to shorten and ease the debugging process (in our case, since we obtain the trees from a formal calculus, we are able to prove the correctness and completeness of the technique), which mitigates the main problem of declarative debugging, the complexity of the questions asked to the user; trusting of statements; `undo` and `don't know` commands; and different strategies to traverse the tree. We refer to [21, 16] for the meaning of these concepts. With respect to other approaches, such as the Maude sufficient completeness checker [6, Chap. 21] or the sets of descendants [8], our tool provides a wider approach, since we handle conditional statements and our equations are not required to be left-linear.

The rest of the paper is structured as follows. Section 2 provides a summary of the main concepts of rewriting logic and Maude specifications. Section 3 describes our calculus and Section 4 shows the debugging trees obtained from it. Finally, Section 5 concludes and mentions some future work.

Detailed proofs of the results can be found in [19], while additional examples, the source code of the tool, and other papers on the subject, including the user guide [16], where a graphical user interface for the debugger is presented, are all available from the webpage `http://maude.sip.ucm.es/debugging`.

[1]Although the least sort error can be seen as a Maude-directed problem, normal forms are a common feature in several programming languages.

2. Rewriting Logic and Maude

Maude modules are executable rewriting logic specifications. Rewriting logic [10] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [2], which, in addition to equations, allows to state of membership axioms characterizing the elements of a sort. Rewriting logic extends *MEL* by adding rewrite rules.

For our purposes in this paper, we are interested in a subclass of rewriting logic models [10] that we call *term models*, where the syntactic structure of terms is kept and associated notions such as variables, substitutions, and term rewriting make sense. These models will be used in Section 4 to represent the *intended interpretation* that the user had in mind while writing a specification. Since we want to find the discrepancies between the intended model and the initial model of the specification as written, we need to consider the relationship between a specification defined by a set of equations E and a set of rules R, and a model defined by possibly different sets of equations E' and of rules R'; in particular, when $E' = E$ and $R' = R$, the term model coincides with the initial model built in [10].

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with Σ a signature, E a set of equations, and R a set of rules, a Σ-term model has an underlying (Σ, E')-algebra whose elements are equivalence classes $[t]_{E'}$ of ground Σ-terms modulo some set of equations and memberships E' (which may be different from E), and there is a transition from $[t]_{E'}$ to $[t']_{E'}$ when $[t]_{E'} \rightarrow^*_{R'/E'} [t']_{E'}$, where rewriting is considered on equivalence classes [10, 7]. The set of rules R' may also be different from R, that is, the term model is $\mathcal{T}_{\Sigma/E',R'}$ for some E' and R'. In such term models, the notion of valuation coincides with that of (ground) substitution. A term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies, under a substitution θ, an equation $u = v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u = v$, when $\theta(u) =_{E'} \theta(v)$, or equivalently, when $[\theta(u)]_{E'} = [\theta(v)]_{E'}$; a membership $u : s$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u : s$, when the Σ-term $\theta(u)$ has sort s according to the information in the signature Σ and the equations and memberships E'; a rewrite $u \Rightarrow v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u \Rightarrow v$, when there is a transition in $\mathcal{T}_{\Sigma/E',R'}$ from $[\theta(u)]_{E'}$ to $[\theta(v)]_{E'}$, that is, when $[\theta(u)]_{E'} \rightarrow^*_{R'/E'} [\theta(v)]_{E'}$. Satisfaction is extended to conditional sentences as usual. A Σ-term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ when $\mathcal{T}_{\Sigma/E',R'}$ satisfies the equations and memberships in E and the rewrite rules in R in this sense. For example, this is obviously the case when $E \subseteq E'$ and $R \subseteq R'$; as mentioned above, when $E' = E$ and $R' = R$ the term model coincides with the initial model for $\mathcal{R}$.

Maude functional modules [6, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); *subsort* relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Maude system modules [6, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

We present how to use this syntax by means of an example. Given a maze, we want to obtain all the possible paths to the exit. First, we define the sorts `Pos`, `List`, and `State` that stand for positions in the labyrinth, lists of positions, and the path traversed so far respectively:

```
(mod MAZE is
  pr NAT .                 sorts Pos List State .
```

Terms of sort `Pos` have the form `[X,Y]`, where `X` and `Y` are natural numbers, and lists are built with `nil` and the juxtaposition operator `_ _`:

```
  subsort Pos < List .       op [_,_] : Nat Nat -> Pos [ctor] .
  op nil : -> List [ctor] . op _ _ : List List -> List [ctor assoc id: nil] .
```

Terms of sort `State` are lists enclosed by curly brackets, that is, `{_}` is an "encapsulation operator" that ensures that the whole state is used:

```
op {_} : List -> State [ctor] .
```

The predicate `isSol` checks whether a list is a solution in a 5×5 labyrinth:

```
vars X Y : Nat .     var P Q : Pos .     var L : List .
op isSol : List -> Bool .
eq [is1] : isSol(L [5,5]) = true .
eq [is2] : isSol(L) = false [owise] .
```

The next position is computed with rule `expand`, that extends the solution with a new position by rewriting `next(L)` to obtain a new position and then checking whether this list is correct with `isOk`. Note that the choice of the next position, that could be initially wrong, produces an implicit backtracking:

```
crl [expand] : { L } => { L P } if next(L) => P /\ isOk(L P) .
```

The function `next` is defined in a nondeterministic way, where `sd` denotes the symmetric difference:

```
op next : List -> Pos .
rl [n1] : next(L [X,Y]) => [X, Y + 1] .
rl [n2] : next(L [X,Y]) => [sd(X, 1), Y] .
rl [n3] : next(L [X,Y]) => [X, sd(Y, 1)] .
```

`isOk(L P)` checks that the position `P` is within the limits of the labyrinth, not repeated in `L`, and not part of the wall by using an auxiliary function `contains`:

```
op isOk : List -> Bool .
eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 5 and Y <= 5
           and not(contains(L, [X,Y])) and not(contains(wall, [X,Y])) .
op contains : List Pos -> Bool .
eq [c1] : contains(nil, P) = false .
eq [c2] : contains(Q L, P) = if P == Q then true else contains(L, P) fi .
```

Finally, we define the `wall` of the labyrinth as a list of positions:

```
op wall : -> List .
eq wall = [2,1] [2,2] [3,2] [2,3] [4,3] [5,3] [1,5] [2,5] [3,5] [4,5] .
endm)
```

Now, we can use the module to search the labyrinth's exit from the position `[1,1]` with the Maude command `search`, but it cannot find any path to escape. We will see in Section 4.1 how to debug it.

3. A Calculus for Missing Answers

We describe in this section a calculus to infer, given a term and some constraints, the *complete* set of reachable terms from this term that fulfill the requirements. The proof trees built with this calculus have nodes that justify why the terms are included in the corresponding sets (positive information) but also nodes that justify why there are no more terms (negative information). These latter nodes are then used in the debugging trees to localize as much as possible the reasons responsible for missing answers. This calculus integrates the calculus to deduce substitutions, normal forms, and least sorts that was presented in [18], and that we reproduce here to give the reader an overall view of debugging of missing answers in Maude specifications. Moreover, these calculi extend the calculus in [17], used to deduce judgments corresponding to oriented equations $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$, and to debug wrong answers. All the results in this paper refer to the

complete calculus comprising these three calculi, and thus we consider this work as the final step in the development of foundations for a complete declarative debugger for Maude.

From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the Maude executability requirements, i.e., E is confluent, terminating, maybe modulo some equational axioms such as associativity and commutativity, and sort-decreasing, while R is coherent with respect to E; see [6] for details. Equations corresponding to the axioms form the set A and the equations in $E - A$ can be oriented from left to right.

We introduce the inference rules used to obtain the set of reachable terms given an initial one, a pattern [6], a condition, and a bound in the number of rewrites. First, the pattern P and the condition $\mathcal{C}$ (that can use variables bound by the pattern) are put together by creating the condition $\mathcal{C}' \equiv P := \circledast \wedge \mathcal{C}$, where $\circledast$ is a "hole" that will be filled by the concrete terms to check if they fulfill both the pattern and the condition. Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*. They correspond to the ones used in Maude modules and are defined as follows:

Definition 3.1. A condition $C_1 \wedge \cdots \wedge C_n$ is *admissible* if, for $1 \leq i \leq n$, C_i is

- an equation $u_i = u'_i$ or a membership $u_i : s$ and $vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or
- a matching condition $u_i := u'_i$, u_i is a pattern and $vars(u'_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or
- a rewrite condition $u_i \Rightarrow u'_i$, u'_i is a pattern and $vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$.

Note that the lefthand side of matching conditions and the righthand side of rewrite conditions can contain extra variables that will be instantiated once the condition is solved.

Definition 3.2. A condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ is *admissible* if $P := t \wedge C_1 \wedge \cdots \wedge C_n$ is admissible for t any ground term.

Definition 3.3. A *kind-substitution*, denoted by κ, is a mapping between variables and terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} \,.\, kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the term it binds.

Definition 3.4. A *substitution*, denoted by θ, is a mapping between variables and terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} \,.\, sort(v_i) \geq ls(t_i)$, that is, the sort of each variable is greater than or equal to the least sort of the term it binds. Note that a substitution is a special type of kind-substitution where each term has the sort appropriate to its variable.

Definition 3.5. Given an atomic condition C, we say that a substitution θ is *admissible for* C if

- C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or
- C is a matching condition $u := u'$ and $vars(u') \subseteq dom(\theta)$, or
- C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.

The calculus presented in this section (in Figures 1–4) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model $\mathcal{T}' = \mathcal{T}_{\Sigma/E',R'}$ defined by equations and memberships E' and by rules R':

- Given a term t and a kind-substitution κ, $\mathcal{T}' \models adequateSorts(\kappa) \rightsquigarrow \Theta$ when either $\Theta = \{\kappa\} \wedge \forall v \in dom(\kappa).\mathcal{T}' \models \kappa[v] : sort(v)$ or $\Theta = \emptyset \wedge \exists v \in dom(\kappa).\mathcal{T}' \not\models \kappa[v] : sort(v)$, where $\kappa[v]$ denotes the term bound by v in κ. That is, when all the terms bound in the kind-substitution κ have the appropriate sort, then κ is a substitution and it is returned; otherwise (at least one of the terms has an incorrect sort), the kind-substitution is not a substitution and the empty set is returned.
- Given an admissible substitution θ for an atomic condition C, $\mathcal{T}' \models [C, \theta] \rightsquigarrow \Theta$ when $\Theta = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \upharpoonright_{dom(\theta)} = \theta\}$, that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ by binding the new variables appearing in C.

- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta \rangle \rightsquigarrow \Theta'$ when $\Theta' = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \upharpoonright_{dom(\theta)} = \theta \text{ for some } \theta \in \Theta\}$, that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.
- $\mathcal{T}' \models \mathit{disabled}(a, t)$ when the equation or membership a cannot be applied to t at the top.
- $\mathcal{T}' \models t \to_{red} t'$ when either $\mathcal{T}' \models t \to^{1}_{E'} t'$ or $\mathcal{T}' \models t_i \to^{!}_{E'} t'_i$, with $t_i \neq t'_i$, for some subterm t_i of t such that $t' = t[t_i \mapsto t'_i]$, that is, the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.
- $\mathcal{T}' \models t \to_{norm} t'$ when $\mathcal{T}' \models t \to^{!}_{E'} t'$, that is, t' is in normal form with respect to the equations E'.
- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t)$ when there exists a substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ holds when $\circledast$ is substituted by t.
- Given an admissible condition $\mathcal{C}$ as before, $\mathcal{T}' \models \mathit{fails}(\mathcal{C}, t)$ when there exists *no* substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.
- $\mathcal{T}' \models t :_{ls} s$ when $\mathcal{T}' \models t : s$ and moreover s is the least sort with this property (with respect to the ordering on sorts obtained from the signature Σ and the equations and memberships E' defining the Σ-term model $\mathcal{T}'$).
- $\mathcal{T}' \models t \Rightarrow^{top} S$ when $S = \{t' \mid t \to^{top}_{R'} t'\}$, that is, the set S is formed by all the reachable terms from t by exactly one rewrite *at the top* with the rules R' defining $\mathcal{T}'$. Moreover, equality in S is modulo E', i.e., we are implicitly working with equivalence classes of ground terms modulo E'.
- $\mathcal{T}' \models t \Rightarrow^{q} S$ when $S = \{t' \mid t \to^{top}_{\{q\}} t'\}$, that is, the set S is the complete set of reachable terms (modulo E') obtained from t with one application of the rule $q \in R'$ at the top.
- $\mathcal{T}' \models t \Rightarrow_1 S$ when $S = \{t' \mid t \to^{1}_{R'} t'\}$, that is, the set S is constituted by all the reachable terms (modulo E') from t in exactly one step, where the rewrite step can take place anywhere in t.
- $\mathcal{T}' \models t \rightsquigarrow^{\mathcal{C}}_{n} S$ when $S = \{t' \mid t \to^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t')\}$, that is, S is the set of all the terms (modulo E') that satisfy the admissible condition $\mathcal{C}$ and are reachable from t in at most n steps.

We first introduce in Figure 1 the inference rules defining the relations $[C, \theta] \rightsquigarrow \Theta$, $\langle C, \Theta \rangle \rightsquigarrow \Theta'$, and $\mathit{adequateSorts}(\kappa) \rightsquigarrow \Theta$. Intuitively, these judgments will provide positive information when they lead to nonempty sets (indicating that the condition holds in the first two judgments or that the kind-substitution is a substitution in the third one) and negative information when they lead to the empty set (indicating respectively that the condition fails or the kind-substitution is not a substitution):

- Rule PatC computes all the possible substitutions that extend θ and satisfy the matching of the term t_2 with the pattern t_1 by first computing the normal form t' of t_2, obtaining then all the possible kind-substitutions κ that make t' and $\theta(t_1)$ equal modulo axioms (indicated by $\equiv_A$), and finally checking that the terms assigned to each variable in the kind-substitutions have the appropriate sort with $\mathit{adequateSorts}(\kappa)$. The union of the set of substitutions thus obtained constitutes the set of substitutions that satisfy the matching.
- Rule AS_1 checks whether the terms of the kind-substitution have the appropriate sort to match the variables. In this case the kind-substitution is a substitution and it is returned.
- Rule AS_2 indicates that, if any of the terms in the kind-substitution has a sort bigger than the required one, then it is not a substitution and thus the empty set of substitutions is returned.
- Rule MbC_1 returns the current substitution if a membership condition holds.

$$\frac{\theta(t_2) \to_{norm} t' \quad adequateSorts(\kappa_1) \rightsquigarrow \Theta_1 \quad \ldots \quad adequateSorts(\kappa_n) \rightsquigarrow \Theta_n}{[t_1 := t_2, \theta] \rightsquigarrow \bigcup_{i=1}^{n} \Theta_i} \mathsf{PatC}$$
if $\{\kappa_1, \ldots, \kappa_n\} = \{\kappa\theta \mid \kappa(\theta(t_1)) \equiv_A t'\}$

$$\frac{t_1 : sort(v_1) \quad \ldots \quad t_n : sort(v_n)}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \{v_1 \mapsto t_1; \ldots; v_n \mapsto t_n\}} \mathsf{AS_1}$$

$$\frac{t_i :_{ls} s_i}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \emptyset} \mathsf{AS_2} \text{ if } s_i \not\leq sort(v_i)$$

$$\frac{\theta(t) : s}{[t : s, \theta] \rightsquigarrow \{\theta\}} \mathsf{MbC_1} \qquad \frac{\theta(t) :_{ls} s'}{[t : s, \theta] \rightsquigarrow \emptyset} \mathsf{MbC_2} \text{ if } s' \not\leq s$$

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \rightsquigarrow \{\theta\}} \mathsf{EqC_1} \qquad \frac{\theta(t_1) \to_{norm} t'_1 \quad \theta(t_2) \to_{norm} t'_2}{[t_1 = t_2, \theta] \rightsquigarrow \emptyset} \mathsf{EqC_2} \text{ if } t'_1 \not\equiv_A t'_2$$

$$\frac{\theta(t_1) \rightsquigarrow_{n+1}^{t_2 := \circledast} S}{[t_1 \Rightarrow t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}} \mathsf{RlC}$$
if $n = min(x \in \mathbb{N} : \forall i \geq 0 \; (\theta(t_1) \rightsquigarrow_{x+i}^{t_2 := \circledast} S))$

$$\frac{[C, \theta_1] \rightsquigarrow \Theta_1 \quad \cdots \quad [C, \theta_m] \rightsquigarrow \Theta_m}{\langle C, \{\theta_1, \ldots, \theta_m\}\rangle \rightsquigarrow \bigcup_{i=1}^{m} \Theta_i} \mathsf{SubsCond}$$

Figure 1: Calculus for substitutions

- Rule $\mathsf{MbC_2}$ is used when the membership condition is not satisfied. It checks that the least sort of the term is not less than or equal to the required one, and thus the substitution does not satisfy the condition and the empty set is returned.
- Rule $\mathsf{EqC_1}$ returns the current substitution when an equality condition holds, that is, when the two terms can be joined with equations, abbreviated as $t_1 \downarrow t_2$.
- Rule $\mathsf{EqC_2}$ checks that an equality condition fails by obtaining the normal forms of both terms and then examining that they are different.
- Rewrite conditions are handled by rule RlC. This rule extends the set of substitutions by computing all the reachable terms that satisfy the pattern (using the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ explained below) and then using these terms to obtain the new substitutions.
- Finally, rule SubsCond computes the extensions of a set of admissible substitutions $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

We use these judgments to define the inference rules of Figure 2, that describe how the normal form and the least sort of a term are computed:

- Rule Dsb indicates when an equation or membership a cannot be applied to a term t. It checks that there are no substitutions that satisfy the matching of the term with the lefthand side of the statement and that fulfill its condition. Note that we check the conditions from left to right, following the same order as Maude and making all the substitutions admissible.
- Rule $\mathsf{Rdc_1}$ reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions. While in the previous rule we made explicit the evaluation from left to right of the condition to show that finally the set of substitutions fulfilling it was empty, in this case we only need one substitution to fulfill the condition and the order is unimportant.
- Rule $\mathsf{Rdc_2}$ reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0 \rangle \rightsquigarrow \Theta_1 \quad \ldots \quad \langle C_n, \Theta_{n-1} \rangle \rightsquigarrow \emptyset}{disabled(a, t)} \ \mathsf{Dsb}$$
$$\text{if } a \equiv l \to r \Leftarrow C_1 \wedge \ldots \wedge C_n \in E \text{ or}$$
$$a \equiv l : s \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^n \quad \{\theta(v_j) : s_j\}_{j=1}^m}{\theta(l) \to_{red} \theta(r)} \ \mathsf{Rdc_1} \ \text{ if } l \to r \Leftarrow \bigwedge_{i=1}^n u_i = u'_i \wedge \bigwedge_{j=1}^m v_j : s_j \in E$$

$$\frac{t \to_{norm} t'}{f(t_1, \ldots, t, \ldots, t_n) \to_{red} f(t_1, \ldots, t', \ldots, t_n)} \ \mathsf{Rdc_2} \ \text{ if } t \not\equiv_A t'$$

$$\frac{disabled(e_1, f(t_1, \ldots, t_n)) \quad \ldots \quad disabled(e_l, f(t_1, \ldots, t_n)) \quad t_1 \to_{norm} t_1 \quad \ldots \quad t_n \to_{norm} t_n}{f(t_1, \ldots, t_n) \to_{norm} f(t_1, \ldots, t_n)} \ \mathsf{Norm}$$
$$\text{if } \{e_1, \ldots, e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1, \ldots, t_n)\}$$

$$\frac{t \to_{red} t_1 \quad t_1 \to_{norm} t'}{t \to_{norm} t'} \ \mathsf{NTr} \qquad \frac{t \to_{norm} t' \quad t' : s \quad disabled(m_1, t') \quad \ldots \quad disabled(m_l, t')}{t :_{ls} s} \ \mathsf{Ls}$$
$$\text{if } \{m_1, \ldots, m_l\} = \{m \in E \mid m \ll_K^{top} t' \ \wedge \ sort(m) < s\}$$

Figure 2: Calculus for normal forms and least sorts

$$\frac{fulfilled(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}} \ \mathsf{Rf_1} \qquad \frac{fails(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset} \ \mathsf{Rf_2}$$

$$\frac{\theta(P) \downarrow t \quad \{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^n \quad \{\theta(v_j) : s_j\}_{j=1}^m \quad \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^l}{fulfilled(\mathcal{C}, t)} \ \mathsf{Fulfill}$$
$$\text{if } \mathcal{C} \equiv P := \circledast \wedge \bigwedge_{i=1}^n u_i = u'_i \wedge \bigwedge_{j=1}^m v_j : s_j \wedge \bigwedge_{k=1}^l w_k \Rightarrow w'_k$$

$$\frac{[P := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0 \rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1} \rangle \rightsquigarrow \emptyset}{fails(\mathcal{C}, t)} \ \mathsf{Fail} \ \text{ if } \mathcal{C} \equiv P := \circledast \wedge C_1 \ \wedge \ \ldots \ \wedge \ C_k$$

Figure 3: Calculus for solutions

- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top considering the variables at the kind level (which is indicated by $\ll_K^{top}$) and that all its subterms are already in normal form.
- Rule NTr describes the transitivity for the reduction to normal form. It reduces the term with the relation $\to_{red}$ and the term thus obtained then is reduced to normal form by using again $\to_{norm}$.
- Rule Ls computes the least sort of the term t. It computes a sort for its normal form (that has the least sort of the terms in the equivalence class) and then checks that memberships deducing lesser sorts, applicable at the top with the variables considered at the kind level, cannot be applied.

In these rules Dsb provides the negative information, proving why the statements (either equations or membership axioms) cannot be applied, while the remaining rules provide the positive information indicating why the normal form and the least sort are obtained.

Once these rules have been introduced, we can use them in the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$. First, we present in Figure 3 the rules related to $n = 0$ steps:

$$\frac{\mathit{fulfilled}(\mathcal{C}, t) \quad t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i \ \cup \ \{t\}} \ \mathsf{Tr_1}$$

$$\frac{\mathit{fails}(\mathcal{C}, t) \quad t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i} \ \mathsf{Tr_2}$$

$$\frac{f(t_1, \ldots, t_m) \Rightarrow^{top} S_t \quad t_1 \Rightarrow_1 S_1 \quad \cdots \quad t_m \Rightarrow_1 S_m}{f(t_1, \ldots, t_m) \Rightarrow_1 S_t \ \cup \ \bigcup_{i=1}^{m} \{f(t_1, \ldots, u_i, \ldots, t_m) \mid u_i \in S_i\}} \ \mathsf{Stp}$$

$$\frac{t \Rightarrow^{q_1} S_{q_1} \quad \cdots \quad t \Rightarrow^{q_l} S_{q_l}}{t \Rightarrow^{top} \bigcup_{i=1}^{l} S_{q_i}} \ \mathsf{Top} \ \text{ if } \{q_1, \ldots, q_l\} = \{q \in R \mid q \ll_K^{top} t\}$$

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0 \rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1} \rangle \rightsquigarrow \Theta_k}{t \Rightarrow^q \bigcup_{\forall \theta \in \Theta_k} \{\theta(r)\}} \ \mathsf{Rl} \ \text{ if } q : l \Rightarrow r \Leftarrow C_1 \ \wedge \ \ldots \ \wedge \ C_k \ \in R$$

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S} \ \mathsf{Red}$$

Figure 4: Calculus for missing answers

- Rule $\mathsf{Rf_1}$ indicates that when only zero steps are used and the current term fulfills the condition, the set of reachable terms consists only of this term.
- Rule $\mathsf{Rf_2}$ complements $\mathsf{Rf_1}$ by defining the empty set as result when the condition does not hold.
- Rule Fulfill checks whether a term satisfies a condition. The premises of this rule check that all the atomic conditions hold, taking into account that it starts with a matching condition with a hole that must be filled with the current term and thus proved with the premise $\theta(P) \downarrow t$. Note that when the condition is satisfied we do not need to check *all* the substitutions, but only to verify that there exists *one* substitution that makes the condition true.
- To check that a term does not satisfy a condition, it is not enough to check that there exists a substitution that makes it to fail; we must make sure that there is no substitution that makes it true. We use the rules shown in Figure 1 to prove that the set of substitutions that satisfy the condition (where the first set of substitutions is obtained from the first matching condition filling the hole with the current term) is empty. Note that, while rule Fulfill provides the positive information indicating that a condition is fulfilled, this one provides negative information, proving that the condition fails.

Now we introduce in Figure 4 the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ when the bound n is greater than 0, which can be understood as searches in *zero or more* steps:

- Rules $\mathsf{Tr_1}$ and $\mathsf{Tr_2}$ show the behavior of the calculus when at least one step can be used. First, we check whether the condition holds (rule $\mathsf{Tr_1}$) or not (rule $\mathsf{Tr_2}$) for the current term, in order to introduce it in the result set. Then, we obtain all the terms reachable in one step with the relation $\Rightarrow_1$, and finally we compute the reachable solutions from these terms

constrained by the same condition and the bound decreased in one step. The union of the sets obtained in this way and the initial term, if needed, corresponds to the final result set.

- Rule Stp shows how the set for one step is computed. The result set is the union of the terms obtained by applying each rule *at the top* (calculated with $t \Rightarrow^{top} S$) and the terms obtained by rewriting one step the arguments of the term. This rule can be straightforwardly adapted to the more general case in which the operator f has some *frozen* arguments (i.e., that cannot be rewritten); the implementation of the debugger makes use of this more general rule.
- How to obtain the terms by rewriting at the top is explained by rule Top, that specifies that the result set is the union of the sets obtained with all the possible applications of each rule of the program. We restrict these rules to those whose lefthand side, with the variables considered at the kind level, matches the term.
- Rule Rl uses the rules in Figure 1 to compute the set of terms obtained with the application of a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed, and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms. The kind of information provided by this rule corresponds to the information provided by the substitutions; if the empty set of substitutions is obtained (negative information) then the rule computes the empty set of terms, which also corresponds with negative information proving that no terms can be obtained with this program rule; analogously when the set of substitutions is nonempty (positive information). This information is propagated through the rest of inference rules justifying why some terms are reachable while others are not.
- Finally, rule Red reduces the reachable terms in order to obtain their normal forms. We use this rule to reproduce Maude behavior, first the normal form is computed and then the rules are applied.

Now we state that this calculus is correct in the sense that the derived judgments with respect to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ coincide with the ones satisfied by the corresponding initial model $\mathcal{T}_{\Sigma/E,R}$, i.e., for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$. This is well known for the judgments corresponding to equations $t = t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$ [11, 10].

Theorem 3.6. *The calculus of Figures 1, 2, 3, and 4 is correct.*

Once these rules are defined, we can build the tree corresponding to the search result shown in Section 2 for the maze example. We recall that we have defined a system to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it. First of all, we have to use a concrete bound to build the tree. It must suffice to compute all the reachable terms, and in this case the least of these values is 4. We have depicted the tree in Figure 5, where we have abbreviated the equational condition `{L:List}` $:= \circledast \wedge$ `isSol(L:List) = true` by $\mathcal{C}$ and `isSol(L) = true` by `isSol(L)`. The leftmost tree justifies that the search condition does not hold for the initial term (this is the reason why Tr_2 has been used instead of Tr_1) and thus it is not a solution. Note that first the substitutions from the matching with the pattern are obtained (`L` $\mapsto$ `[1,1]` in this case), and then these substitutions are used to instantiate the rest of the condition, that for this term does not hold, which is proved by $*_1$. The next tree shows the set of reachable terms in one step (the tree $*_2$, explained below, computes the terms obtained by rewrites at the top, while the tree on its right shows that the subterms cannot be further rewritten) and finally the rightmost tree, that has a similar structure to this one and will not be studied in depth, continues the search with a decreased bound.

The tree $*_1$ shows why the current list is not a solution (i.e., the tree provides the negative information proving that this fragment of the condition does not hold). The reason is that the term `isSol(L)` is reduced to `false`, when we needed it to be reduced to `true`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{1} \rightarrow_{norm} \mathtt{1}}\,\mathsf{Norm}}{\mathtt{[1,1]} \rightarrow_{norm} \mathtt{[1,1]}}\,\mathsf{Norm}}{\mathtt{\{[1,1]\}} \rightarrow_{norm} \mathtt{\{[1,1]\}}}\,\mathsf{Norm}}{[\mathtt{\{L:List\}} := \mathtt{\{[1,1]\}}, \emptyset] \rightsquigarrow \mathtt{L} \mapsto \mathtt{[1,1]}}\,\mathsf{PatC} \quad \dfrac{*_1}{\langle \mathtt{isSol(L)}, \{\mathtt{L} \mapsto \mathtt{[1,1]}\}\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{\mathit{fails}(\mathcal{C}, \mathtt{\{[1,1]\}})}\,\mathsf{Fail} \quad \dfrac{\dfrac{*_2 \quad \dfrac{\dfrac{}{\mathtt{[1,1]} \Rightarrow^{top} \emptyset}\,\mathsf{Top} \quad \dfrac{\dfrac{}{\mathtt{1} \Rightarrow^{top} \emptyset}\,\mathsf{Top}}{\mathtt{1} \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{\mathtt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{\mathtt{\{[1,1]\}} \Rightarrow_1 \mathtt{\{[1,1][1,2]\}}}\,\mathsf{Stp} \quad \dfrac{*_3}{\mathtt{\{[1,1][1,2]\}} \rightsquigarrow^{\mathcal{C}}_3 \emptyset}\,\mathsf{Tr}_2}{\mathtt{\{[1,1]\}} \rightsquigarrow^{\mathcal{C}}_4 \emptyset}\,\mathsf{Tr}_2$$

Figure 5: Tree for the maze example

$$\dfrac{\dfrac{\dfrac{}{\mathtt{isSol([1,1])} \rightarrow_{red} \mathtt{false}}\,\mathsf{Rdc}_1 \quad \dfrac{}{\mathtt{false} \rightarrow_{norm} \mathtt{false}}\,\mathsf{Norm}}{\mathtt{isSol([1,1])} \rightarrow_{norm} \mathtt{false}}\,\mathsf{NTr} \quad \dfrac{}{\mathtt{true} \rightarrow_{norm} \mathtt{true}}\,\mathsf{Norm}}{[\mathtt{isSol(L)\ =\ true}, \mathtt{L} \mapsto \mathtt{[1,1]}] \rightsquigarrow \emptyset}\,\mathsf{EqC}_2$$

Figure 6: Tree $*_1$ for the search condition

The tree labeled with $*_2$ is sketched in Figure 7. In this tree the applications of all the rules whose lefthand side matches the current term (`{[1,1]}`) are tried. In this case only the rule `expand` (abbreviated by `e`) can be used, and it generates a list with the new position `[1,2]`; the tree $*_4$ is used to justify that the first condition of `expand` holds and extends the set of substitutions that fulfill the condition thus far to the set $\{\theta_1, \theta_2, \theta_3\}$, where $\theta_1 \equiv \mathtt{L} \mapsto \mathtt{[1,1]};\mathtt{P} \mapsto \mathtt{[1,2]}$, $\theta_2 \equiv \mathtt{L} \mapsto \mathtt{[1,1]};\mathtt{P} \mapsto \mathtt{[1,0]}$, and $\theta_3 \equiv \mathtt{L} \mapsto \mathtt{[1,1]};\mathtt{P} \mapsto \mathtt{[0,1]}$. The substitution θ_1 also fulfills the next condition, `isOk(L P)`, which is proved with the rule EqC_1 in ♣ (where the big triangle is a computation in the calculus of [17] proving that the conditions of the equations hold), while the substitutions θ_2 and θ_3 fail; the trees $\bigtriangledown$ proving it are analogous to the one shown in Figure 6. This substitution θ_1 is thus the only one inferred in the root of the tree, where the node ♣ provides the positive information proving why the substitution is obtained and its siblings ($\bigtriangledown$) the negative information proving why the other substitutions are not in the set.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{1} \rightarrow_{norm} \mathtt{1}}\,\mathsf{Norm}}{\mathtt{\{[1,1]\}} \rightarrow_{norm} \mathtt{\{[1,1]\}}}\,\mathsf{Norm}}{[\mathtt{\{L\}} := \mathtt{\{[1,1]\}}, \emptyset] \rightsquigarrow \{\mathtt{L} \mapsto \mathtt{[1,1]}\}}\,\mathsf{PatC} \quad *_4 \quad \dfrac{\dfrac{\dfrac{\bigtriangledown}{\mathtt{isOk([1,1][1,2])} \rightarrow \mathtt{true}}\,\mathsf{Rep}_{\rightarrow} \quad \dfrac{}{\mathtt{true} \rightarrow \mathtt{true}}\,\mathsf{Rf}_{\rightarrow}}{(\clubsuit)\ [\mathtt{isOk(L\ P)}, \theta_1] \rightsquigarrow \{\theta_1\}}\,\mathsf{EqC}_1 \quad \bigtriangledown \quad \bigtriangledown}{\langle \mathtt{isOk(L\ P)}, \{\theta_1, \theta_2, \theta_3\}\rangle \rightsquigarrow \{\theta_1\}}\,\mathsf{SubsCond}}{\mathtt{\{[1,1]\}} \Rightarrow^{\mathtt{e}} \mathtt{\{[1,1][1,2]\}}}\,\mathsf{Rl}}{\mathtt{\{[1,1]\}} \Rightarrow^{top} \mathtt{\{[1,1][1,2]\}}}\,\mathsf{Top}$$

Figure 7: Tree $*_2$ for the applications at the top

The tree $*_4$, shown in Figure 8, is in charge of inferring the set of substitutions obtained when checking the first condition of the rule `expand`, namely `next(L) => P`. The condition is instantiated with the substitution obtained from matching the term with the lefthand side of the rule (in this case $\mathtt{L} \mapsto \mathtt{[1,1]}$) and, since it is a rewrite condition, the set of reachable terms—computed with $*_5$, which will not be further discussed here—is used to extend this substitution, obtaining a set with three different substitutions (that we previously abbreviated as θ_1, θ_2, and θ_3).

4. Debugging Trees

We describe in this section how to obtain appropriate debugging trees from the proof trees introduced in the previous section. Following the approach shown in [17], we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user *expects* that $\mathcal{I} \models \varphi$ for any judgment φ deduced with respect to the

$$\dfrac{\dfrac{\dfrac{*_5}{\texttt{next([1,1])} \rightsquigarrow_2^{\texttt{P}:=\circledast} \{\texttt{[1,2]},\texttt{[1,0]},\texttt{[0,1]}\}}\ \mathsf{Tr}_2}{[\texttt{next(L)} \Rightarrow \texttt{P}, \texttt{L} \mapsto \texttt{[1,1]}] \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,2]},\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,0]},\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[0,1]}\}}\ \mathsf{RIC}}{\langle \texttt{next(L)} \Rightarrow \texttt{P}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,2]},\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,0]},\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[0,1]}\}}\ \mathsf{SubsCond}$$

Figure 8: Tree $*_4$ for the first condition of `expand`

rewrite theory $\mathcal{R}$. We will say that a judgment is *valid* when it holds in the intended interpretation $\mathcal{I}$, and *invalid* otherwise. Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy: If all the children of N are valid, then finish pointing out at N as buggy; otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node. Proving that this strategy is complete is straightforward by using induction on the height of T. By using the proof trees computed with the calculus of the previous section as debugging trees we are able to locate wrong statements, missing statements, and wrong search conditions, which are defined as follows:

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$, a membership $l : s$, or a rule $l \Rightarrow r$) and a substitution θ, the *statement instance* $\theta(A) \Leftarrow \theta(C_1) \wedge \cdots \wedge \theta(C_n)$ is *wrong* when all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$ but $\theta(A)$ is not.
- Given a rule $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$ and a term t, the rule has a *wrong instance* if the judgments $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, $\cdots$, $[C_n, \Theta_{n-1}] \rightsquigarrow \Theta_n$ are valid in $\mathcal{I}$ but the application of Θ_n to the righthand side does not provide all the results expected for this rule.
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, $\cdots$, $[C_n, \Theta_{n-1}] \rightsquigarrow \emptyset$ are valid in $\mathcal{I}$ (meaning that the condition does not hold for t) but the user expected the condition to hold, then we have a *wrong search condition instance*.
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if there exists a substitution θ such that $\theta(l) \equiv_A t$ and all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$, but the condition is not expected to hold, then we also have a *wrong search condition instance*.
- A statement or condition is *wrong* when it admits a wrong instance.
- Given a term t, there is a *missing equation for* t if the computed normal form of t does not correspond with the one expected in $\mathcal{I}$. A specification has a *missing equation* if there exists a term t such that there is a missing equation for t.
- Given a term t, there is a *missing membership for* t if the computed least sort for t does not correspond with the one expected in $\mathcal{I}$. A specification has a *missing membership* if there exists a term t such that there is a missing membership for t.
- Given a term t, there is a *missing rule for* t if all the rules applied to t at the top lead to judgments $t \Rightarrow^{q_i} S_{q_i}$ valid in $\mathcal{I}$ but the union $\bigcup S_{q_i}$ does not contain all the reachable terms from t by using rewrites at the top. A specification has a *missing rule* if there exists a term t such that there is a missing rule for t.[2]

In our debugging framework, when a wrong statement is found, this specific statement is pointed out; when a missing statement is found, the debugger indicates the operator at the top of the term in the lefthand side of the statement that is missing; and when a wrong condition is found, the specific condition is shown. We will see in the next section that some extra information will be kept in the tree to provide this information. It is important not to confuse missing answers with missing

[2]Actually, what the debugger reports is that a statement is missing *or* the conditions in the remaining statements are not the intended ones (thus they are not applied when expected and another one would be needed), but the error *is not located* in the statements used in the conditions, since they are also checked during the debugging process.

statements; the current calculus detects missing answers due to both wrong and missing statements and wrong search conditions.

4.1. Abbreviated Proof Trees

We will not use proof trees T directly as debugging trees, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT when T is clear from the context. The reason for preferring the APT is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. This transformation relies on the following proposition:

Proposition 4.1. *Let N be a buggy node in some proof tree in the calculus of Figures 1, 2, 3, and 4, w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N is the consequence of a* Rep$_{\rightarrow}$, Rep$_{\Rightarrow}$, Mb, Rdc$_1$, Norm, Ls, Fulfill, Fail, Top, *or* Rl *inference rule.*
2. *The error associated to N is a wrong statement, a missing statement, or a wrong search condition.*

To indicate the error associated to the buggy node, we assume that the nodes inferred with these inference rules are decorated with some extra information to identify the error when they are pointed out as buggy. More specifically, nodes related to wrong statements keep the label of the statement, nodes related to missing statements keep the operator at the top that requires more statements to be defined, and nodes related to wrong conditions keep the condition.

The key idea in the APT, whose rules are shown in Figure 9, is to keep the nodes related to the inference rules indicated in Proposition 4.1, making use of the rest of rules to improve the questions asked to the user. The abbreviation always starts by applying ($\mathbf{APT}_1$). This rule simply duplicates the root of the tree and applies APT', which receives a proof tree and returns a forest (i.e., a set of trees). Hence without this duplication the result of the abbreviation could be a forest instead of a single tree. The rest of the APT rules correspond to the function APT' and are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule of least number. The following advantages are obtained:

- Questions associated to nodes with reductions are improved (rules ($\mathbf{APT}_2$), ($\mathbf{APT}_3$), ($\mathbf{APT}_5$), ($\mathbf{APT}_6$), and ($\mathbf{APT}_7$)) by asking about normal forms instead of asking about intermediate states. For example, in rule ($\mathbf{APT}_2$) the error associated to $t \rightarrow t'$ is the one associated to $t \rightarrow t''$, which is not included in the APT. We have chosen to introduce $t \rightarrow t'$ instead of simply $t \rightarrow t''$ in the APT as a pragmatic way of simplifying the structure of the APTs, since t' is obtained from t'' and hence likely simpler.
- The rule ($\mathbf{APT}_4$) deletes questions about rewrites *at the top* (that can be difficult to answer due to matching modulo) and associates the information of those nodes to questions related to the set of reachable terms in one step with rewrites in any position, that are in general easier to answer.
- It creates, with the variants of the rules ($\mathbf{APT}_8$) and ($\mathbf{APT}_9$), two different kinds of tree, one that contains judgments of rewrites with several steps and another that only contains rewrites in one step. The one-step debugging tree follows strictly the idea of keeping only nodes corresponding to relevant information. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rules. The user will choose which debugging tree (one-step or many-steps) will be used for the debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites gives to some parts of the debugging tree the shape of a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to use

$$(\mathbf{APT}_1) \quad APT\left(\frac{T_1 \ldots T_n}{aj}\, R_1\right) = \frac{APT'\left(\frac{T_1 \ldots T_n}{aj}\, R_1\right)}{aj}$$

$$(\mathbf{APT}_2) \quad APT'\left(\frac{\frac{T_1 \ldots T_n}{t \to t''}\,\mathsf{Rep}_{\to} \quad T'}{t \to t'}\,\mathsf{Tr}_{\to}\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)\ APT'(T')}{t \to t'}\,\mathsf{Rep}_{\to}\right\}$$

$$(\mathbf{APT}_3) \quad APT'\left(\frac{\frac{T_1 \ \ldots \ T_n}{t \to t''}\,\mathsf{Rdc}_1 \ T'}{t \to t'}\,\mathsf{NTr}\right) = \left\{\frac{APT'(T_1) \ \ldots \ APT'(T_n)\ APT'(T')}{t \to t'}\,\mathsf{Rdc}_1\right\}$$

$$(\mathbf{APT}_4) \quad APT'\left(\frac{\frac{T_1 \ \ldots \ T_n}{t \Rightarrow^{top} S'}\,\mathsf{Top}\ T'_1 \ldots T'_m}{t \Rightarrow_1 S}\,\mathsf{Stp}\right) = \left\{\frac{APT'(T_1) \ \ldots \ APT'(T_n)\ APT'(T'_1) \ \ldots \ APT'(T'_m)}{t \Rightarrow_1 S}\,\mathsf{Top}\right\}$$

$$(\mathbf{APT}_5) \quad APT'\left(\frac{T' \quad \frac{T_1 \ldots T_n}{t \Rightarrow t'}\,\mathsf{Rep}_{\Rightarrow} \quad T''}{t_1 \Rightarrow t_2}\,\mathsf{EC}\right) = \left\{\frac{APT'(T')\ APT'(T_1) \ldots APT'(T_n)\ APT'(T'')}{t_1 \Rightarrow t_2}\,\mathsf{Rep}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}_6) \quad APT'\left(\frac{T \quad \frac{T_1 \ldots T_n}{aj'}\, R_1 \quad T'}{aj}\,\mathsf{Red}\right) = \left\{\frac{APT'(T)\ APT'(T_1) \ \ldots \ APT'(T_n)\ APT'(T)}{aj}\, R_1\right\}$$

$$(\mathbf{APT}_7) \quad APT'\left(\frac{T_{t \to_{norm} t'}\ T_1 \ldots T_n}{t :_{ls} s}\,\mathsf{Ls}\right) = \left\{\frac{APT'(T_{t \to_{norm} t'})\ APT'(T_1) \ \ldots \ APT'(T_n)}{t' :_{ls} s}\,\mathsf{Ls}\right\}$$

$$(\mathbf{APT}^o_8) \quad APT'\left(\frac{T_1 \quad T_2}{t \Rightarrow t'}\,\mathsf{Tr}_{\Rightarrow}\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}^m_8) \quad APT'\left(\frac{T_1 \quad T_2}{t \Rightarrow t'}\,\mathsf{Tr}_{\Rightarrow}\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t \Rightarrow t'}\,\mathsf{Tr}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}^o_9) \quad APT'\left(\frac{T_1 \ldots T_n}{aj}\,\mathsf{Tr}\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

$$(\mathbf{APT}^m_9) \quad APT'\left(\frac{T_1 \ldots T_n}{aj}\,\mathsf{Tr}_i\right) = \left\{\frac{APT'(T_1) \ \ldots \ APT'(T_n)}{aj}\,\mathsf{Tr}_i\right\}$$

$$(\mathbf{APT}_{10}) \quad APT'\left(\frac{T_1 \ldots T_n}{aj}\, R_2\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)}{aj}\, R_2\right\}$$

$$(\mathbf{APT}_{11}) \quad APT'\left(\frac{T_1 \ldots T_n}{aj}\, R_1\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

R_1 any inference rule R_2 either Mb, $\mathsf{Rep}_{\to}$, $\mathsf{Rep}_{\Rightarrow}$, Rdc_1, Norm, Fulfill, Fail, Ls, Rl, or Top

$1 \leq i \leq 2$ aj, aj' any judgment

Figure 9: Transforming rules for obtaining abbreviated proof trees

efficiently the divide and query navigation strategy. On the contrary, removing the transitivity inferences for rewrites (as rules $(\mathbf{APT}^o_8)$ and $(\mathbf{APT}^o_9)$ do) produces flattened trees where this strategy is no longer efficient. On the other hand, in rewrites $t \Rightarrow t'$ and searches $t \leadsto^{\mathcal{C}}_n S$ appearing as conclusion of a transitivity inference rule, the judgment can be more complicated because it combines several inferences. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{(\spadesuit)\ \texttt{1} \to_{norm} \texttt{1}}\ \mathsf{Norm}_{\texttt{s_}}}{(\spadesuit)\ \texttt{[1,1]} \to_{norm} \texttt{[1,1]}}\ \mathsf{Norm}_{\texttt{[_,_]}}}{(\spadesuit)\ \texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\ \mathsf{Norm}_{\texttt{\{_\}}} \quad \cfrac{}{\texttt{isSol}(P_1) \to \texttt{f}}\ \mathsf{Rdc}_{\texttt{is2}} \quad \star_1 \quad \triangledown \quad \ldots \quad \triangledown \quad \star_2}{\texttt{\{[1,1]\}} \leadsto_4^{\mathcal{C}} \emptyset}\ \mathsf{Tr}_2$$

Figure 10: Abbreviated proof tree for the maze example

- The rule ($\mathbf{APT}_{11}$) removes from the tree all the nodes not associated with relevant information, since the rule ($\mathbf{APT}_{10}$) keeps the relevant information and the rules are applied in order. We remove, for example, nodes related to judgments about sets of substitutions, disabled statements, and rewrites with a concrete rule, that can be in general difficult to answer. Moreover, it removes from the tree trivial judgments like the ones related to reflexivity or congruence.
- Since the APT is built without computing the associated proof tree, it reduces the time and space needed to build the tree.

The following theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and functional modules: the technique will find a buggy node starting from any initial symptom. We assume that the information introduced by the user during the session is correct.

Theorem 4.2. *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 2, 3, and 4 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ s.t. the root of T is invalid in $\mathcal{I}$. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ (correctness).*

The trees in Figures 10–12 depict the (one-step) abbreviated proof tree for the maze example, where $\mathcal{C}$ stands for `{L:List}:=` $\circledast$ $\wedge$ `isSol(L:List)`, P_1 for `[1,1]`, L_1 for `[1,1][1,2]`, L_2 for `[1,1][1,0]`, L_3 for `[1,1][0,1]`, `t` for `true`, `f` for false, `n` for `next`, `e` for `expand`, L for `[1,1][1,2][1,3][1,4]`, and $*_5'$ for the application of APT' to $*_5$. We have also extended the information in the labels with the operator or statement associated to the inference. More concretely, the tree in Figure 10 abbreviates the tree in Figure 5; the first two premises in the abbreviated tree abbreviate the first premise in the proof tree (which includes the tree in Figure 6), keeping only the nodes associated with relevant information according to Proposition 4.1: Norm, with the operator associated to the reduction, and Rdc_1, with the label of the associated equation. The tree $\star_1$, shown in Figure 11, abbreviates the second premise of the tree in Figure 5 as well as the trees in Figures 7 and 8; it only keeps the nodes referring to normal forms, searches in one step, that are now associated to the rule Top, each of them referring to a different operator (the operator `s_` is the successor constructor for natural numbers), and the applications of rules (Rl) and equations ($\mathsf{Rep}_\to$). Note that the equation describing the behavior of `isOk` has not got any label, which is indicated with the symbol $\perp$; we will show below how the debugger deals with these nodes. The tree $\star_2$, presented in Figure 12, shares these characteristics and only keeps nodes related to one-step searches and application of rules.

These APT rules are combined with trusting mechanisms that further reduce the proof tree (note that the correctness of these techniques relies on the decisions made by the user):

- Statements can be trusted in several ways: non labelled statements are always trusted (i.e., the nodes marked with ($\diamondsuit$) in Figure 11 will be discarded by the debugger), statements and modules can be trusted before starting the debugging process, and statements can also be trusted on the fly.

$$\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \texttt{1} \to_{norm} \texttt{1}}\,\mathsf{Norm}_{\mathsf{s_}}}{(\spadesuit)\ \texttt{[1,1]} \to_{norm} \texttt{[1,1]}}\,\mathsf{Norm}_{\mathsf{[_,_]}} \quad \dfrac{*'_5 \quad \dfrac{}{(\Diamond)\ \texttt{isOk}(L_1) \to \texttt{t}}\,\mathsf{Rep}_\bot \quad \dfrac{}{(\Diamond)\ \texttt{isOk}(L_2) \to \texttt{f}}\,\mathsf{Rep}_\bot \quad \dfrac{}{(\Diamond)\ \texttt{isOk}(L_3) \to \texttt{f}}\,\mathsf{Rep}_\bot}{\texttt{\{[1,1]\}} \Rightarrow^{\mathsf{e}} \texttt{\{[1,1][1,2]\}}}\,\mathsf{Rl}_{\mathsf{e}} \quad \dfrac{\dfrac{}{(\heartsuit)\ \texttt{1} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathsf{s_}}}{(\heartsuit)\ \texttt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathsf{[_,_]}}}{\texttt{\{[1,1]\}} \Rightarrow_1 \texttt{\{[1,1][1,2]\}}}\,\mathsf{Top}_{\mathsf{\{_\}}}$$

Figure 11: Abbreviated tree $\star_1$

$$\dfrac{\dfrac{\dfrac{\dfrac{\bigtriangledown \quad \cdots \quad \bigtriangledown}{\texttt{n}(L) \Rightarrow^{\texttt{n1}} \texttt{[1,5]}}\,\mathsf{Rl}_{\texttt{n1}} \quad \dfrac{\bigtriangledown \quad \cdots \quad \bigtriangledown}{\texttt{n}(L) \Rightarrow^{\texttt{n2}} \texttt{[0,4]}}\,\mathsf{Rl}_{\texttt{n2}} \quad \dfrac{\bigtriangledown \quad \cdots \quad \bigtriangledown}{\texttt{n}(L) \Rightarrow^{\texttt{n3}} \texttt{[1,3]}}\,\mathsf{Rl}_{\texttt{n3}}}{(\ddagger)\ \texttt{n}(L) \Rightarrow_1 \{\texttt{[1,5]}, \texttt{[0,4]}, \texttt{[1,3]}\}}\,\mathsf{Top}_{\texttt{n}} \quad \bigtriangledown \quad \cdots \quad \bigtriangledown}{(\dagger)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow^{\mathsf{e}} \emptyset}\,\mathsf{Rl}_{\mathsf{e}}}{(\dagger)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathsf{\{_\}}}$$

Figure 12: Abbreviated tree $\star_2$

- A correct module can be given before starting a debugging session. By checking the correctness of the judgments against this module, correct nodes can be deleted from the tree.
- We consider that constructed terms (terms built only with constructors, pointed out with the `ctor` attribute, and also known as data terms in other contexts) are in normal form and thus inferences of the form $t \to_{norm} t$ with t constructed are removed from the tree. This would remove from the tree the nodes marked with ($\spadesuit$) in Figures 10 and 11.
- Constructed terms of certain sorts or built with some operators can be considered *final*, which indicates that they cannot be further rewritten. For example, we could consider terms of sorts `Nat` and `List` to be final and thus the nodes marked with ($\heartsuit$) in Figure 11 would be removed from the tree.

Once this tree has been built, we can use it to debug the error shown in Section 2. Using the top-down navigation strategy our tool would show all the children of the root and ask the user to select an incorrect one. The last one (the root of $\star_2$) is incorrect and can be selected, and then the user has to answer about the validity of the child of this node. Since it is also incorrect the debugger selects it as current one (the path thus far has been marked with (†) in Figure 12) and the debugger shows its children. The first child (‡) is erroneous, but this time its children are all correct, so the tool points it out as buggy and it is associated to an erroneous fragment of code. More concretely, the rule used to infer this judgment was Top, and it is associated with the operator `next` (that was abbreviated as `n`), i.e., another rule for this operator is needed. Indeed, if we check the module we notice that the movement to the right has not been specified. We can fix it by adding:

```
rl [n4] : next(L [X,Y]) => [X + 1, Y] .
```

A detailed session of this example is available in the webpage `maude.sip.ucm.es/debugging`.

5. Conclusions and Future Work

We have presented in this paper a debugger of missing answers for Maude specifications. The trees for this kind of debugging are obtained from an abbreviation of a proper calculus whose adequacy for debugging has been proved. This work extends our previous work on wrong and missing answers [17, 18] and provides a powerful and complete debugger for Maude specifications. Moreover, we also provide a graphical user interface that eases the interaction with the debugger and improves its traversal. The tree construction, its navigation, and the user interaction (excluding the GUI) have been all implemented in Maude itself. For more information, see `http://maude.sip.ucm.es/debugging`.

We plan to add new navigation strategies like the ones shown in [21] that take into account the number of different potential errors in the subtrees, instead of their size. Moreover, the current

version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session. We intend to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. Finally, we also plan to create a test generator to test Maude specifications and debug the erroneous test with the debugger.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05), Tallinn, Estonia*, pages 8–13. ACM Press, 2005.

[4] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In F. J. López-Fraguas, editor, *15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.

[5] R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative diagnosis of missing answers in constraint functional-logic programming. In J. Garrigue and M. V. Hermenegildo, editors, *Proceedings of 9th International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2008.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990.

[8] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA 98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[9] I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[11] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[12] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.

[13] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[14] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[15] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

[16] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009. `http://maude.sip.ucm.es/debugging`.

[17] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of rewriting logic specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques (WADT 2008)*, volume 5486 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2009.

[18] A. Riesco, A. Verdejo, and N. Martí-Oliet. Enhancing the debugging of Maude specifications. In *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA 2010)*, Lecture Notes in Computer Science, 2010. To appear.

[19] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. Technical Report SIC 02/10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2010. http://maude.sip.ucm.es/debugging.

[20] E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation. MIT Press, 1983.

[21] J. Silva. A comparative study of algorithmic debugging strategies. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[22] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 295-310
http://rewriting.loria.fr/rta/

SIMULATION IN THE CALL-BY-NEED LAMBDA-CALCULUS WITH LETREC

MANFRED SCHMIDT-SCHAUSS [1] AND DAVID SABEL [1] AND ELENA MACHKASOVA [2]

[1] Dept. Informatik und Mathematik, Inst. Informatik, Goethe-University, PoBox 11 19 32, D-60054 Frankfurt, Germany
E-mail address, M. Schmidt-Schauß: `schauss@ki.informatik.uni-frankfurt.de`
E-mail address, D. Sabel: `sabel@ki.informatik.uni-frankfurt.de`

[2] Division of Science and Mathematics, University of Minnesota, Morris, MN 56267-2134, U.S.A
E-mail address, E. Machkasova: `elenam@morris.umn.edu`

Abstract. This paper shows the equivalence of applicative similarity and contextual approximation, and hence also of bisimilarity and contextual equivalence, in the deterministic call-by-need lambda calculus with letrec. Bisimilarity simplifies equivalence proofs in the calculus and opens a way for more convenient correctness proofs for program transformations. Although this property may be a natural one to expect, to the best of our knowledge, this paper is the first one providing a proof. The proof technique is to transfer the contextual approximation into Abramsky's lazy lambda calculus by a fully abstract and surjective translation. This also shows that the natural embedding of Abramsky's lazy lambda calculus into the call-by-need lambda calculus with letrec is an isomorphism between the respective term-models. We show that the equivalence property proven in this paper transfers to a call-by-need letrec calculus developed by Ariola and Felleisen.

1. Introduction

Non-strict programming languages such as the core-language of Haskell can be modeled using call-by-need lambda calculi. Contextual semantics, based on an operational semantics, describes behavior of expressions in all possible contexts and can model the semantics of different variants of these calculi. Applicative bisimulation is a restricted form of contextual equivalence: if two closed expressions behave the same on all arguments, then they are bisimilar. It allows more convenient proofs of e.g. correctness of program transformations. Abramsky & Ong showed that applicative bisimulation is the same as contextual equivalence in a specific simple lazy lambda calculus [Abr90, Abr93], and Howe [How89, How96] proved that in classes of calculi applicative bisimulation is the same as contextual equivalence. This leads to the expectation that some form of applicative bisimulation may be used for calculi with Haskell's cyclic let(rec). Howe's method is applicable to calculi with non-recursive let even in the presence of non-determinism [Man10]. However, in the case of (cyclic) letrec

1998 ACM Subject Classification: F.4.2, F.3.2, F.3.3, F.4.1.
Key words and phrases: semantics, contextual equivalence, bisimulation, lambda calculus, call-by-need, letrec.

© M. Schmidt-Schauß, D. Sabel, and E. Machkasova
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.295

and non-determinism the method fails, as a recent counterexample shows [SS09a]. This raises a question: which call-by-need calculi with letrec permit applicative bisimilarity as a tool for proving contextual equality.

We show in this paper that for the minimal extension of Abramsky's lazy lambda calculus with letrec which implements sharing and explicit recursion, the equivalence of contextual equivalence and applicative bisimulation indeed holds. The technique used is via two translations: W from a call-by-need letrec-calculus into a full call-by-name letrec calculus using infinite trees as justification for the correctness (i.e. full abstraction), and N translating the letrec expressions away using a family of fixpoint combinators. Full abstraction of the translation, an analysis of applicative contexts, and a variant of behavioral similarity then show that the applicative similarity can be transferred between the calculi and that the embedding of the lazy lambda calculus into the call-by-need calculus is an isomorphism of the respective term models.

In [Jef94] there is an investigation into the semantics of a lambda-calculus that permits cyclic graphs, and where a fully abstract denotational semantics is described. However, the calculus is different from our calculi in its expressiveness since it permits strictness annotations and a parallel convergence test, where the latter is required for the full abstraction property of the denotational model. Expressiveness of programming languages was investigated e.g. in [Fel91] and the usage of syntactic methods was formulated as a research program there, with non-recursive let as the paradigmatic example. Our isomorphism-theorem 6.9 shows that this approach is extensible to a cyclic let.

Related work on calculi with recursive bindings includes the following foundational papers. An early paper that proposes cyclic let-bindings (as graphs) is [Ari94], where reduction and confluence properties are discussed. [Ari95, Ari97, Mar98] present call-by-need lambda calculi with non-recursive let and a let-less formulation of call-by-need reduction. For a calculus with non-recursive let it is shown in [Mar98] that call-by-name and call-by-need evaluation induce the same observational equivalences. Call-by-need lambda calculi with a recursive let that closely correspond to our calculus L_{need} are also presented in [Ari95, Ari97, Ari02]. In [Ari02] it is shown that there exist infinite normal forms and that the calculus satisfies a form of confluence. In this paper we show that the letrec calculus of [Ari97] is equivalent to L_{need} w.r.t. convergence and contextual equivalence (see Theorem 7.1) and that bisimulation for the letrec calculus of [Ari97] is equivalent to contextual equivalence. This supports our experience and view that contextual equivalence is a more central notion than a specific standard reduction.

Outline: In Sect. 3 we introduce the two letrec-calculi and recall results for Abramsky's lazy lambda calculus. In Sect. 4 and 5 the translations W and N are introduced and the full-abstraction results are obtained. In Sect. 6 we show that bisimulation and contextual equivalence are the same in the call-by-need calculus with letrec. In Sect. 7 we show that our result is transferable to the letrec-calculus of [Ari97]. Finally, we conclude in Sect. 8.

2. Common Notions and Notations for Calculi

Before we explain the specific calculi, some common notions are introduced. A calculus definition consists of its syntax together with its operational semantics which defines the evaluation of programs and the implied equivalence of expressions.

Definition 2.1. An untyped deterministic *calculus* $\mathcal{D}$ is a four-tuple $(\mathcal{E}, \mathcal{C}, \to, \mathcal{W})$, where $\mathcal{E}$ are expressions, $\mathcal{C} : \mathcal{E} \to \mathcal{E}$ is a set of functions (which usually represents contexts), $\to$

is a small-step reduction relation (usually the normal-order reduction), which is a partial function on expressions, and $\mathcal{W} \subset \mathcal{E}$ is a set of *values* of the calculus.

For $C \in \mathcal{C}$ and an expression s, the functional application is denoted as $C[s]$. For contexts, this is the replacement of the hole of C by s. We also assume that the identity function Id is contained in $\mathcal{C}$ with $Id[s] = s$ for all expressions s.

The *transitive closure* of $\rightarrow$ is denoted as $\xrightarrow{+}$ and the *transitive and reflexive closure* of $\rightarrow$ is denoted as $\xrightarrow{*}$. Given an expression t, a sequence $t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ is called a *reduction sequence*; it is called an *evaluation* if t_n is a value, i.e. $t_n \in W$. Then we say s *converges* and denote this as $s{\downarrow}t_n$ or as $s{\downarrow}$ if t_n is not important. If there is no t_n s.t. $s{\downarrow}t_n$ then s *diverges*, denoted as $s{\Uparrow}$. When dealing with multiple calculi, we often use the calculus name to mark its expressions and relations, e.g. $\xrightarrow{\mathcal{D}}$ denotes a reduction relation in $\mathcal{D}$.

Contextual approximation and equivalence can be defined in a general way:

Definition 2.2. Let $D = (\mathcal{E}, \mathcal{C}, \rightarrow, \mathcal{W})$ be a calculus and s, t be D-expressions. *Contextual approximation* $\leq_D$ and *contextual equivalence* $\sim_D$ are defined as:

$$\begin{array}{lll} s \leq_D t & \text{iff} & \forall C \in \mathcal{C}: \ C[s]{\downarrow}_D \Rightarrow C[t]{\downarrow}_D \\ s \sim_D t & \text{iff} & s \leq_D t \wedge t \leq_D s \end{array}$$

Note that $\leq_D$ is a precongruence and that $\sim_D$ is a congruence.

We are interested in translations between calculi that are faithful w.r.t. the corresponding contextual preorders. Recall that we developed such translations between calculi with contextual equivalences in [SS08b, SS09b]: A translation $\tau : (\mathcal{E}_1, \mathcal{C}_1, \rightarrow_1, \mathcal{W}_1) \rightarrow (\mathcal{E}_2, \mathcal{C}_2, \rightarrow_2, \mathcal{W}_2)$ is a mapping $\tau_E : \mathcal{E}_1 \rightarrow \mathcal{E}_2$ and a mapping $\tau_C : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ such that $\tau_C(Id_1) = Id_2$. The following notions are defined:

- τ is *compositional* iff $\tau(C[e]) = \tau(C)[\tau(e)]$ for all C, e.
- τ is *convergence equivalent* iff $e{\downarrow}_1 \iff \tau(e){\downarrow}_2$ for all e.
- τ is *adequate* iff for all $e, e' \in \mathcal{E}_1$: $\tau(e) \sim_2 \tau(e') \implies e \sim_1 e'$.
- τ is *fully abstract* iff for all $e, e' \in \mathcal{E}_1$: $e \sim_1 e' \iff \tau(e) \sim_2 \tau(e')$.

From [SS08b, SS09b] it is known that a compositional and convergence equivalent translation is adequate.

3. Three Calculi

In this section we present the calculi that we use in the paper: the two calculi L_{need} and L_{name} with letrec, which have the same syntax, but differ in their reduction strategies, and Abramsky's "lazy lambda calculus", which is a pure lambda calculus with a call-by-name reduction that has abstractions as successful results.

3.1. The Call-by-Need Calculus L_{need}

We begin with the call-by-need lambda calculus L_{need} which is exactly the call-by-need calculus of [SS07]. The set $\mathcal{E}$ of L_{need}-expressions is as follows where x, x_i are variables:

$$s_i, s, t \in \mathcal{E} \ ::= \ x \mid (s\ t) \mid (\lambda x.s) \mid (\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$$

We assign the names *application*, *abstraction*, or `letrec`-*expression* to the expressions $(s\ t)$, $(\lambda x.s)$, $(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ t)$, respectively. A group of `letrec` bindings is abbreviated as Env.

(lbeta)	$C[((\lambda x.s)^S\ r)] \to C[(\texttt{letrec}\ x = r\ \texttt{in}\ s)]$
(cp-in)	$(\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V]) \to (\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[s])$ where s is an abstraction or a variable
(cp-e)	$(\texttt{letrec}\ x = s^S, Env, y = C[x^V]\ \texttt{in}\ r) \to (\texttt{letrec}\ x = s, Env, y = C[s]\ \texttt{in}\ r)$ where s is an abstraction or a variable
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^S) \to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^S\ \texttt{in}\ r)$ $\to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$C[((\texttt{letrec}\ Env\ \texttt{in}\ t)^S\ s)] \to C[(\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))]$

Figure 1: Reduction rules of L_{need}

We assume that variables x_i in `letrec`-bindings are all distinct, that letrec-expressions are identified up to reordering of binding-components, and that, for convenience, there is at least one binding. `letrec`-bindings are recursive, i.e., the scope of x_j in ($\texttt{letrec}\ x_1 = s_1, \ldots, x_{n-1} = s_{n-1}\ \texttt{in}\ s_n$) are all expressions s_i with $1 \leq i \leq n$. Free and bound variables in expressions and α-renamings are defined as usual. The set of free variables in t is denoted as $FV(t)$. We use the distinct variable convention, i.e., all bound variables in expressions are assumed to be distinct, and free variables are distinct from bound variables. The reduction rules are assumed to implicitly α-rename bound variables in the result if necessary.

A *context* C is an expression from L_{need} extended by a symbol $[\cdot]$, the *hole*, such that $[\cdot]$ occurs exactly once (as subexpression) in C. Given a term t and a context C, we write $C[t]$ for the L_{need}-expression constructed from C by plugging t into the hole, i.e, by replacing $[\cdot]$ in C by t, where this replacement is meant syntactically, i.e., a variable capture is permitted.

Definition 3.1. The *reduction rules* for the calculus and language L_{need} are defined in Fig. 1, where the labels S, V are used for the exact definition of the normal-order reduction below. Several reduction rules are denoted by their name prefix, e.g. the union of (llet-in) and (llet-e) is called (llet). The union of (llet) and (lapp) is called (lll).

For the definition of the normal order reduction strategy of the calculus L_{need} we use the labeling algorithm in Figure 2, which detects the position to which a reduction rule is applied according to the normal order. It uses the following labels: S (subterm), T (top term), V (visited). We use $\vee$ when a rule allows two options for a label, e.g. $s^{S \vee T}$ stands for s labeled with S or T. A labeling rule $l \to r$ is applicable to a (labeled) expression s if s matches l with the labels given by l where s may have more labels than l if not otherwise stated. The labeling algorithm has as input an expression s and then exhaustively applies the rules in Fig. 2 to s^T, where no other subexpression in s is labeled. The label T is used to prevent the labeling algorithm from visiting `letrec`-environments that are not at the top of the expression. The labeling algorithm either terminates with *fail* or with success, where in general the direct superterm of the S-marked subexpression indicates a potential normal-order redex. The use of such a labeling algorithm corresponds to the search of a redex in term graphs where it is usually called unwinding.

Example 3.2. For the expression $\texttt{letrec}\ x = x\ \texttt{in}\ x$ the labeling does not fail:

$$(\texttt{letrec}\ x = x\ \texttt{in}\ x)^T \to (\texttt{letrec}\ x = x\ \texttt{in}\ x^S)^V \to (\texttt{letrec}\ x = x^S\ \texttt{in}\ x^V)^V$$

$$\begin{array}{lll}
(\texttt{letrec}\ Env\ \texttt{in}\ t)^T & \rightarrow & (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V \\
C[(s\ t)^{S \vee T}] & \rightarrow & C[(s^S\ t)^V] \\
(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) & \rightarrow & (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V]) \\
(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ t) & \rightarrow & (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ t) \\
 & & \text{if } s \text{ was not labeled and if } C[x] \neq x \\
(\texttt{letrec}\ x = s^V, y = C[x^S], Env\ \texttt{in}\ t) & \rightarrow & \textit{fail} \text{ if } C[x] \neq x \\
(\texttt{letrec}\ x = C[x^S]^V, Env\ \texttt{in}\ t) & \rightarrow & \textit{fail} \text{ if } C[x] \neq x
\end{array}$$

Figure 2: Labeling algorithm for L_{need}

But for the expressions $\texttt{letrec}\ x = (y\ x), y = (x\ y)\ \texttt{in}\ x$ and $\texttt{letrec}\ x = (x\ \lambda u.u)\ \texttt{in}\ x$ the labeling fails.

Definition 3.3 (Normal Order Reduction of L_{need})**.** Let t be an expression. Then a single normal order reduction step $\xrightarrow{need}$ is defined as follows: first the labeling algorithm is applied to t. If the labeling algorithm terminates successfully, then one of the rules in Figure 1 is applied, if possible, where the labels S, V must match the labels in the expression t (again t may have more labels). The *normal order redex* is defined as the left-hand side of the applied reduction rule. The notation for a normal-order reduction that applies the rule a is $\xrightarrow{need,a}$, e.g. $\xrightarrow{need,lapp}$ applies the rule $(lapp)$.

Definition 3.4. A *reduction context* R_{need} is any context, such that its hole is labeled with S or T by the labeling algorithm.

Note that the normal order redex as well as the normal order reduction is unique. A *weak head normal form in* L_{need} *(*L_{need}*-WHNF)* is either an abstraction $\lambda x.s$, or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ \lambda x.s)$. The notions of convergence, divergence and contextual approximation are as defined in Sect. 2. Note that black holes, i.e. expressions with cyclic dependencies in a normal order reduction context, diverge, e.g. $\texttt{letrec}\ x = x\ \texttt{in}\ x$. Other expressions which diverge are open expressions where a free variable appears (perhaps after several reductions) in reduction position. A specific representative of diverging expressions is $\Omega := (\lambda z.(z\ z))\ (\lambda x.(x\ x))$, i.e. $\Omega\Uparrow_{need}$.

Example 3.5. We consider the expression $t_1 := \texttt{letrec}\ x = (y\ \lambda u.u), y = \lambda z.z\ \texttt{in}\ x$. The labeling algorithm applied to t_1 yields $(\texttt{letrec}\ x = (y^V\ \lambda u.u)^V, y = (\lambda z.z)^S\ \texttt{in}\ x^V)^V$. The only reduction rule that matches this labeling is the reduction rule (cp-e), i.e. $t_1 \xrightarrow{need} (\texttt{letrec}\ x = ((\lambda z'.z')\ \lambda u.u), y = (\lambda z.z)\ \texttt{in}\ x) = t_2$. The labeling of t_2 is $(\texttt{letrec}\ x = ((\lambda z'.z')^S\ \lambda u.u)^V, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$, which makes the reduction (lbeta) applicable, i.e. $t_2 \xrightarrow{need} (\texttt{letrec}\ x = (\texttt{letrec}\ z' = \lambda u.u\ \texttt{in}\ z'), y = (\lambda z.z)\ \texttt{in}\ x) = t_3$. The labeling of t_3 is $(\texttt{letrec}\ x = (\texttt{letrec}\ z' = \lambda u.u\ \texttt{in}\ z')^S, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$. Thus an (llet-e)-reduction is applicable to t_2, i.e. $t_3 \xrightarrow{L_{need}} (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ x) = t_4$. Application of the labeling algorithm to t_4 yields: $(\texttt{letrec}\ x = z'^S, z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ x^V)^V$. Thus the normal order reduction is a (cp-in)-reduction, i.e. $t_4 \xrightarrow{L_{need}} (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ z') = t_5$ The labeling of t_5 is $(\texttt{letrec}\ x = z', z' = \lambda u.u^S, y = (\lambda z.z)\ \texttt{in}\ z'^V)^V$. Again a (cp-e) reduction is applicable, i.e. $t_5 \rightarrow (\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ \lambda u'.u') = t_6$ The labeling algorithm applied to t_6

yields $(\texttt{letrec}\ x = z', z' = \lambda u.u, y = (\lambda z.z)\ \texttt{in}\ \lambda u'.u'^S)^V$, but no reduction is applicable to t_6, since t_6 is a WHNF.

3.2. The Call-by-Name Calculus L_{name}

Now we define a call-by-name calculus on the L_{need}-syntax. The syntax of the calculus L_{name} is the same as that of L_{need}, but the reduction rules are different. This calculus L_{name} has a different call-by-name-reduction than the one in [SS07], since that calculus treats only beta-redexes as call-by-name, but uses a sharing variant for (cp).

The reduction contexts R_{name} are contexts of the form $L[A]$ where the context classes $\mathcal{A}$ and $\mathcal{L}$ are defined by $L \in \mathcal{L} ::= [\cdot] \mid \texttt{letrec}\ \mathit{Env}\ \texttt{in}\ L$; $A \in \mathcal{A} ::= [\cdot] \mid (A\ s)$ where s is any expression. Normal order reduction $\xrightarrow{name}$ is defined by the following three rules:

(lapp)	$R_{name}[(\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)\ s]$	$\rightarrow R_{name}[\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ (t\ s)]$
(beta)	$R_{name}[((\lambda x.s)\ t)]$	$\rightarrow R_{name}[s[t/x]]$
(cp)	$L[\texttt{letrec}\ \mathit{Env},\ x = s\ \texttt{in}\ R_{name}[x]]$	$\rightarrow L[\texttt{letrec}\ \mathit{Env},\ x = s\ \texttt{in}\ R_{name}[s]]$

Note that $\xrightarrow{name}$ is unique. An L_{name}-WHNF is defined as an expression of the form $L[\lambda x.s]$. We write $s{\downarrow}_{name}$ iff there is a normal-order reduction to a L_{name}-WHNF, i.e. iff $s \xrightarrow{name,*} L[\lambda x.s']$.

3.3. The Lazy Lambda Calculus

In this subsection we give a short description of the lazy lambda calculus [Abr90], denoted with L_{lazy}, which is a call-by-name lambda calculus. The set $\mathcal{E}$ of L_{lazy}-expressions is that of the usual (untyped) lambda calculus: $s, s_i, t \in \mathcal{E} ::= x \mid (s_1\ s_2) \mid (\lambda x.s)$ where e, e_i are expressions, and x means a variable. The set $\mathcal{W}$ of *values* are the L_{lazy}-abstractions. The reduction contexts $\mathcal{R}_{lazy}$ are defined by $R_{lazy} \in \mathcal{R}_{lazy} := [\cdot] \mid (R_{lazy}\ s)$ where s is any L_{lazy}-expression. A $\xrightarrow{lazy}$-reduction is defined by the rule: (beta) $R_{lazy}[((\lambda x.s)\ t)] \rightarrow R_{lazy}[s[t/x]]$. The $\xrightarrow{lazy}$-reduction is unique.

We repeat the definitions and the required properties of L_{lazy}, where proofs can be found in [How89, How96, Abr90, Abr93]. For basic definitions and confluence see e.g. [Bar84]. Since this calculus is well-studied and some properties are folklore, there are different and alternative proofs of the properties below. We require these properties in other sections and as properties of the target of translations, which allows us to lift the properties to the calculi L_{name} and L_{need}.

Definition 3.6 (Simulation in L_{lazy})**.** Let η be a binary relation on closed L_{lazy}-expressions. Then $s\ [\eta]_{lazy}\ t$ holds iff $s{\downarrow}\lambda x.s'$ implies $\big(\ t{\downarrow}\lambda x.t'$ and for all closed L_{lazy}-expressions r the relation $s'[r/x]\ \eta\ t'[r/x]$ holds$\big)$. The relation $\leq_{b,lazy}$ is defined as the greatest fixpoint of the operator $[\cdot]_{lazy}$.

For a relation η on closed expressions, let the open extension η^o be defined as $s\ \eta^o\ t$ iff for all closing substitutions σ: $\sigma(s)\ \eta\ \sigma(t)$. Note that by the theorem below, this can be shown to be equivalent to: for all closing substitutions σ that replace variables by closed abstractions or Ω: $\sigma(s)\ \eta\ \sigma(t)$. As an example $\leq^o_{b,lazy}$ is the open extension of $\leq_{b,lazy}$.

There are several variants of behaviorally and contextually defined relations in L_{lazy}, that are all equivalent to contextual approximation.

Theorem 3.7. *In L_{lazy}, all the following relations are equivalent to contextual approximation $\leq_{lazy}$:*

1. $\leq^o_{b,lazy}$.
2. *The relation $\leq_{lazy,1}$ where $s \leq_{lazy,1} t$ iff for all closing contexts C: $C[s]\downarrow \implies C[t]\downarrow$.*
3. *The relation $\leq_{lazy,2}$, defined as: $s \leq_{lazy,2} t$ iff for all closed contexts C and all closing substitutions: $C[\sigma(s)]\downarrow \implies C[\sigma(t)]\downarrow$.*
4. *The relation $\leq^o_{b,lazy,1}$ where $\leq_{b,lazy,1}$ is defined using the Kleene-construction: $\leq_{b,lazy,1} = \bigcap_{i \geq 0} \leq'_{b,i}$, where $\leq'_{b,0}$ is the relation $\mathcal{E} \times \mathcal{E}$, and $\leq'_{b,i+1} := [\leq'_{b,i}]_{lazy}$ for all i.*
5. *The relation $\leq^o_{b,lazy,2}$ where $\leq_{b,lazy,2}$ is defined as: $s \leq_{b,lazy,2} t$ iff for all $n \geq 0$ and all closed expressions $r_i, i = 1, \ldots, n$: $s\ r_1 \ldots r_n\downarrow \implies t\ r_1 \ldots r_n\downarrow$.*
6. *The relation $\leq^o_{b,lazy,3}$, where $\leq_{b,lazy,3}$ is defined as: $s \leq_{b,lazy,3} t$ iff for all $n \geq 0$ and all $r_i, i = 1, \ldots, n$, where r_i may be a closed abstraction or Ω: $s\ r_1 \ldots r_n\downarrow \implies t\ r_1 \ldots r_n\downarrow$.*
7. *The relation $\leq^o_{b,lazy,4}$, where $\leq_{b,lazy,4}$ is the greatest fixpoint of the operator $[\cdot]_{lazy,a\Omega}$ on closed expressions. By definition $s\ [\eta]_{lazy,a\Omega}\ t$ holds iff $s\downarrow\lambda x.s'$ implies $t\downarrow\lambda x.t'$ and for all closed L_{lazy}-abstractions r and $r = \Omega$, the relation $s'[r/x]\ \eta\ t'[r/x]$ holds.*

Beta-reduction is a correct program transformation in L_{lazy}:

Theorem 3.8. *Let s, t be L_{lazy}-expressions. If $s \xrightarrow{beta} t$, then $s \sim_{lazy} t$. For all L_{lazy}-expressions s, t: $\Omega \leq_{lazy} s$. If s, t are closed and $s\Uparrow$ and $t\Uparrow$, then $s \sim_{lazy} t$.*

Also the following can easily be derived from Theorem 3.7 and Theorem 3.8.

Proposition 3.9. *For open L_{lazy}-expressions s, t, where all free variables of s, t are in $\{x_1, \ldots, x_n\}$: $s \leq_{lazy} t \iff \lambda x_1, \ldots x_n.s \leq_{lazy} \lambda x_1, \ldots x_n.t$*

Proposition 3.10. *Given any two closed L_{lazy}-expressions s, t: for all closed L_{lazy}-abstractions r and also for $r = \Omega$ $s\ r \leq_{lazy} t\ r \iff s \leq_{lazy} t$.*

Proof. The if-direction follows from the congruence property. The only-if direction follows from Theorem 3.7. ∎

4. The Translation $W : L_{need} \to L_{name}$

The translation $W : L_{need} \to L_{name}$ is defined as the identity on expressions and contexts, but the convergence predicates are changed. We will prove that contextual equivalence based on L_{need}-evaluation and contextual equivalence based on L_{name}-evaluation are equivalent. We will use infinite trees to connect both evaluation strategies. Note that [SS07] already shows that infinite tree convergence is equivalent to call-by-need convergence. Thus, we mainly treat call-by-name evaluation in this section.

We recall the definition of an infinite tree from [SS07], and describe the set of trees as a calculus in the sense of Section 2 called L_{tree}: The set of infinite trees $\mathcal{T}$ is co-inductively defined using the grammar $T \in \mathcal{T} ::= x \mid (T_1\ T_2) \mid \lambda x.T \mid \bot$ where x is a variable, T, T_1, T_2 are infinite trees, $\bot$ is a (special) constant. Contexts are trees with exactly one occurrence of a hole (as a subexpression).

Definition 4.1. Tree reduction contexts $\mathcal{R}$ for (infinite) trees are inductively defined by $\mathcal{R} ::= [\cdot] \mid (\mathcal{R}\ T)$, where T stands for an infinite tree. The only reduction on trees is:

$$\text{(betaTr)} \qquad ((\lambda x.s)\ r) \to s[r/x]$$

If the reduction rule is applied in an $\mathcal{R}$-context, it is a *normal order reduction on trees* $\xrightarrow{tree}$. Values are trees of the form $\lambda x.T$, i.e. abstractions.

Now we define a translation IT from L_{name}-expressions into L_{tree}-expressions.

We use Dewey notation, i.e. strings over $\{1,2\}$, as positions of infinite trees, where numbers are separated by a period. Here 1 refers to the left and 2 to the right subtree of an application, and 1 to the body of an abstraction. The empty string is denoted as ε. For an infinite tree T its *label at position* p (written as $T\lfloor_p$) is defined as usual, i.e. $(T_1\ T_2)\lfloor_{1.p} = T_1\lfloor_p$, $(T_1\ T_2)\lfloor_{2.p} = T_2\lfloor_p$, $(\lambda x.T)\lfloor_\varepsilon = \lambda x$, $(T_1\ T_2)\lfloor_\varepsilon = app$, $x\lfloor_\varepsilon = x$, and $\bot\lfloor_\varepsilon = \bot$. The subtree of T at position p is $T|_p$.

Definition 4.2. Given an expression t, the infinite tree $IT(t)$ of t is defined by the labels at valid positions, where the positions and the labels of $IT(t)$ for every position are computed by the following algorithm, using the notation $C[t'\downarrow_p]$ if the algorithm searches the label at position p and is currently at the subexpression t'. Given the expression t and a position p, if and only if the following rules ($\mapsto$) (where C, C_i are L_{name}-contexts, s,t are L_{name}-expressions) exhaustively applied to $t\downarrow_p$ end with a label $l \in \{\lambda x, app, x, \bot\}$, then p is a position of $IT(t)$ and $IT(t)\lfloor_p = l$.

The final steps in the label computation are as follows:

$$\begin{array}{lll}
C[(\lambda x.s)\downarrow_\varepsilon] & \mapsto & \lambda x \\
C[(s\ t)\downarrow_\varepsilon] & \mapsto & app \\
C[x\downarrow_\varepsilon] & \mapsto & x \quad \text{if } x \text{ is a free or a lambda-bound variable} \\
C[\texttt{letrec}\ x = C[x\downarrow_\varepsilon], Env\ \texttt{in}\ s] & \mapsto & \bot \\
C[\texttt{letrec}\ x_1 = C_1[y_1], \ldots, x_n = C_n[x_1\downarrow_\varepsilon], Env\ \texttt{in}\ s] & \mapsto & \bot
\end{array}$$

For the general cases, we proceed as follows:

$$\begin{array}{llll}
1. & C[(\lambda x.s)\downarrow_{1.p}] & \mapsto & C[\lambda x.(s\downarrow_p)] \\
2. & C[(s\ t)\downarrow_{1.p}] & \mapsto & C[(s\downarrow_p\ t)] \\
3. & C[(s\ t)\downarrow_{2.p}] & \mapsto & C[(s\ t\downarrow_p)] \\
4. & C[(\texttt{letrec}\ Env\ \texttt{in}\ r)\downarrow_p] & \mapsto & C[(\texttt{letrec}\ Env\ \texttt{in}\ r\downarrow_p)] \\
5. & C_1[(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x\downarrow_p])] & \mapsto & C_1[(\texttt{letrec}\ x = s\downarrow_p, Env\ \texttt{in}\ C_2[x])] \\
6. & C_1[\texttt{letrec}\ x = s, y = C_2[x\downarrow_p], Env\ \texttt{in}\ r] & \mapsto & C_1[\texttt{letrec}\ x = s\downarrow_p, y = C_2[x], Env\ \texttt{in}\ r]
\end{array}$$

In all cases not mentioned above, the result is undefined, and hence the position p is not a position of the tree.

Lemma 4.3. *Let $s,t \in L_{name}$. Then $s \xrightarrow{name,cp} t$ or $s \xrightarrow{name,lapp} t$ implies $IT(s) = IT(t)$.*

Proof. For (cp) let $s = C_1[\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[x]]$ and $t = C_1[\texttt{letrec}\ x = s, Env\ \texttt{in}\ C_2[s]]$. Then for $IT(s)$ and $IT(t)$ the only change may happen at the position that corresponds to x in $C_2[x]$, but as the computation of the labels shows, the labels remain unchanged.

For (lapp) let $s = C[(\texttt{letrec}\ Env\ \texttt{in}\ s')\ t']$ and $t = C[\texttt{letrec}\ Env\ \texttt{in}\ (s'\ t')]$. Then it is again easy to observe that every label of every position is identical for $IT(s)$ and $IT(t)$. ∎

Lemma 4.4. *Let* $s_1 := R_{name}[(\lambda x.s)\ t] \xrightarrow{name,beta} R_{name}[s[t/x]] =: s_2$. *Then* $IT(s_1) \xrightarrow{tree} IT(s_2)$.

Proof. The redex $((\lambda x.s)\ t)$ is mapped by IT to a unique tree position within a tree reduction context in $IT(s_1)$. The computation IT transforms $((\lambda x.s)\ t)$ into a subtree $\sigma((\lambda x.s)\ t)$, where σ is a substitution replacing variables by infinite trees. The tree reduction replaces $\sigma((\lambda x.s)\ t)$ by $\sigma(s)[\sigma(t)/x]$, hence the lemma holds. ∎

Proposition 4.5. *Let* s *be an expression with* $s\downarrow_{name}$. *Then* $IT(s)\downarrow_{tree}$.

Proof. This follows by induction on the length of a normal order reduction of s. The base case holds, since $IT(L[(\lambda x.s)])$ is always a value tree. For the induction step we consider the first reduction of s, say $s \to s'$. The induction hypothesis shows $IT(s')\downarrow_{tree}$. If the reduction $s \to s'$ is a ($name$,lapp) or ($name$,cp) reduction, then Lemma 4.3 implies $IT(s)\downarrow_{tree}$. If $s \xrightarrow{name,beta} s'$, then Lemma 4.4 shows $IT(s) \xrightarrow{tree} IT(s')$ and thus $IT(s)\downarrow_{tree}$. ∎

Now we show the other direction:

Lemma 4.6. *Let* s *be an expression such that* $IT(s) = \mathcal{R}[T]$, *where* $\mathcal{R}$ *is a tree reduction context and* $T \neq \bot$. *Then there is an expression* s' *such that* $s \xrightarrow{name,(lapp)\vee(cp),*} s'$, $IT(s') = IT(s)$, $s' = R[s'']$, $IT(L[s'']) = T$, *where* $R = L[A[\cdot]]$ *is a reduction context for some* $\mathcal{L}$*-context* L *and some* $\mathcal{A}$*-context* A, s'' *is a free variable, an abstraction or an application iff* T *is a free variable, an abstraction or an application, respectively, and the position* p *of the hole in* $\mathcal{R}$ *is also the position of the hole in* $A[\cdot]$.

Proof. The tree T may be an abstraction, an application, or a free variable in $R[T]$. Let p be the position of the hole of $\mathcal{R}$. We will show by induction on the label-computation for p in s that there is a reduction $s \xrightarrow{name,(lapp)\vee(cp),*} s'$, where s' as claimed in the lemma.
We consider the label-computation for p to explain the induction measure, where we use the rule numbers of Definition 4.2. Let q be such that the label computation for p is of the form 4^*q and q does not start with 4. The measure for induction is a tuple (a, b), where a is the length of q, and $b \geq 0$ is the maximal number with $q = 2^b q'$. The base case is (a, a): Then the label computation is of the form 2^* and indicates that s is of the form $L[A[s'']]$ and satisfies the claim of the lemma. For the induction step we have to check several cases:

1. The label computation is of the form $4^*2^+4\ldots$. Then a normal-order (lapp) can be applied to s resulting in s_1. The label-computation for p w.r.t. s_1 is of the same length, and only applications of 2 and 4 are interchanged, hence the second component of the measure is strictly decreased.
2. The label computation is of the form $4^*2^*5\ldots$. Then a normal-order (cp) can be applied to s resulting in s_1. The length q is strictly decreased by 1, and perhaps one 6.-step is changed into a 5.-step. Hence the measure is strictly reduced. ∎

Lemma 4.7. *Let* s *be an expression with* $IT(s) \xrightarrow{tree} T$. *Then there is some* s' *with* $s \xrightarrow{name,*} s'$ *and* $IT(s') = T$.

Proof. If $IT(s) \xrightarrow{tree} T$, then $IT(s) = \mathcal{R}[(\lambda x.t_1)\ t_2]$ where $\mathcal{R}$ is a reduction context and $T = \mathcal{R}[t_1[t_2/x]]$. Let p be the position of the hole of $\mathcal{R}$ in $IT(s)$. We first apply Lemma 4.6 to s and the tree context $\mathcal{R}[([\cdot]\ t_2)]$ and thus obtain a reduction $s \xrightarrow{name,*} s'$, such that $IT(s) = IT(s')$ and $s' = R[r]$ where $R = L[A[\cdot]]$ is a reduction context and $IT(L[r]) = (\lambda x.t_1)$, and

r is an abstraction. It is obvious that $IT(s')|_{p.2} = t_2$ and that $R = L[A'[[\cdot]\ r_2]]$. Thus $s' = L[A'[((\lambda x.r_1)\ r_2)]] \xrightarrow{name,beta} L[A'[r_1[r_2/x]] = s''$. Now one can verify that $IT(s'') = T$ must hold. ∎

Proposition 4.8. *Let s be an expression with $IT(s){\downarrow_{tree}}$. Then $s{\downarrow_{name}}$.*

Proof. We use induction on the length k of a tree reduction $IT(s) \xrightarrow{tree,k} T$, where T is a value tree. For the base case it is easy to verify that if $IT(s)$ is a value tree, then $s \xrightarrow{name,cp,*} L[\lambda x.s']$ for some $\mathcal{L}$-context and some s'. I.e. $s\ {\downarrow_{name}}$. The induction step follows by Lemma 4.7. ∎

Corollary 4.9. *For all L_{name}-expressions s: $s{\downarrow_{name}}$ if, and only if $IT(s){\downarrow_{tree}}$.*

Theorem 4.10. $\leq_{name} = \leq_{need}$

Proof. We have shown that L_{name}-convergence is equivalent to infinite tree convergence. In [SS07] it was shown that L_{need}-convergence is equivalent to infinite tree convergence. Hence, L_{name}-convergence and L_{need}-convergence are equivalent, which also implies that both contextual preorders and also the contextual equivalences are identical. ∎

Corollary 4.11. *W is convergence equivalent and fully abstract.*

5. Translation $N : L_{name} \to L_{lazy}$

We use multi-fixpoint combinators as defined in [Gol05] to translate letrec-expressions into equivalent ones without a `letrec`. The translated expressions belong to L_{lazy}.

Definition 5.1. Given $n > 1$, a family of n fixpoint combinators Y_i^n for $i = 1, \ldots, n$ can be defined as follows:

$$\begin{array}{lll} Y_i^n & := & \lambda f_1, \ldots, f_n.(\ (\lambda x_1, \ldots, x_n.f_i\ \ (x_1\ x_1\ \ldots x_n)\ \ldots\ (x_n\ x_1\ \ldots x_n)) \\ & & \qquad (\lambda x_1, \ldots, x_n.f_1\ \ (x_1\ x_1\ \ldots x_n)\ \ldots\ (x_n\ x_1\ \ldots x_n)) \\ & & \qquad \ldots \\ & & \qquad (\lambda x_1, \ldots, x_n.f_n\ \ (x_1\ x_1\ \ldots x_n)\ \ldots\ (x_n\ x_1\ \ldots x_n))) \end{array}$$

The idea of the translation is to replace (`letrec` $x_1 = s_1, \ldots, x_n = s_n$ `in` r) by $r[S_1/x_1, \ldots, S_n/x_n]$ where $S_i := Y_i^n\ F_1 \ldots F_n$ and $F_i := \lambda x_1, \ldots, x_n.s_i$.

In this way the fixpoint combinators implement the generalized fixpoint property: $Y_i^n\ F_1 \ldots F_n \sim F_i\ (Y_1^n\ F_1 \ldots F_n) \ldots (Y_n^n\ F_1 \ldots F_n)$. However, our translation uses modified expressions, as shown below.

Consider the expression $Y_i^n F_1 \ldots F_n$. Expanding the notations, we get $((\lambda f_1, \ldots, f_n.(X_i\ \ X_1\ \ \ldots\ \ X_n))\ \ F_1\ \ \ldots\ \ F_n)$ where $X_i = \lambda x_1 \ldots x_n.(f_i\ (x_1\ x_1\ \ldots\ x_n)\ \ldots\ (x_n\ x_1\ \ldots\ x_n))$. Reducing further:

$$\begin{array}{l} (\lambda f_1, \ldots, f_n.(X_i\ X_1\ \ldots\ X_n))\ F_1\ \ldots\ F_n \xrightarrow{\beta,*} (X_i'\ X_1'\ \ldots\ X_n'), \\ \text{where } X_i' = \lambda x_1 \ldots x_n.(F_i\ (x_1\ x_1\ \ldots\ x_n) \ldots (x_n\ x_1\ \ldots x_n)) \end{array}$$

We take the latter expression as the definition of the multi-fixpoint translation, where we avoid substitutions and instead generate β-redexes.

Definition 5.2. The translation $N :: L_{name} \to L_{lazy}$ is recursively defined as:

- $N(\texttt{letrec}\ x_1 = s_1, \ldots, x_n = s_n\ \texttt{in}\ r) = ((\lambda x_1 .\ldots x_n.(N(r)))\ U_1\ \ldots\ U_n)$
  where $U_i = (\lambda x_1, \ldots, x_n.x_i\ x_1 \ldots x_n)\ X'_1\ \ldots\ X'_n,$
  $X'_i = \lambda x_1 \ldots x_n.F_i(x_1 x_1 \ldots x_n) \ldots (x_n x_1 \ldots x_n),$
  $F_i = \lambda x_1, \ldots, x_n.N(s_i).$
- $N(s_1\ s_2) = (N(s_1)\ N(s_2))$
- $N(\lambda x.s) = \lambda x.N(s)$
- $N(x) = x.$

We extend N to contexts by treating the hole as a constant, i.e. $N([\cdot]) = [\cdot]$.

Convergence equivalence of the translation N follows by inspecting the relation between L_{name}- and the translated L_{lazy}-reductions. The full proof can be found in [SS10]

Proposition 5.3. *N is convergence equivalent, i.e. $\forall t \in L_{name}$: $t{\downarrow_{name}} \iff N(t){\downarrow_{lazy}}$.*

Lemma 5.4. *The translation N is compositional, i.e. for all expressions t and all contexts C: $N(C[t]) = N(C)[N(t)]$.*

Proof. This easily follows by structural induction on the definition. ∎

Proposition 5.5. *For all $s, t \in L_{name}$: $N(s) \leq_{lazy} N(t) \implies s \leq_{name} t$, i.e. N is adequate.*

Proof. Since N is convergence equivalent (Proposition 5.3) and compositional by Lemma 5.4, we derive that N is adequate (see [SS08b] and Section 2). ∎

Lemma 5.6. *For `letrec`-free expressions s, t of L_{name} the following holds: $s, t \in L_{lazy}$ and $s \leq_{name} t \implies s \leq_{lazy} t$.*

Proof. Clearly every `letrec`-free expression of L_{name} is also an L_{lazy} expression. Let s, t be `letrec`-free such that $s \leq_{name} t$. Let C be an L_{lazy}-context such that $C[s]{\downarrow_{lazy}}$, i.e. $C[s] \xrightarrow{lazy,k} \lambda x.s'$. By comparing the reduction strategies in L_{name} and L_{lazy}, we obtain that $C[s] \xrightarrow{name,k} \lambda x.s'$ (by the identical reduction sequence), since $C[s]$ is `letrec`-free. Thus, $C[s]{\downarrow_{name}}$ and also $C[t]{\downarrow_{name}}$, i.e. there is a normal order reduction in L_{name} for $C[t]$ to a WHNF. Since $C[t]$ is `letrec`-free, we can perform the identical reduction in L_{lazy} and obtain $C[t]{\downarrow_{lazy}}$. ∎

The language L_{lazy} is embedded into L_{name} (and also L_{need}) by the identity embedding $\iota(s) = s$. In the following proposition we show that every L_{need}-WHNF (and also every L_{name}-WHNF) is contextually equivalent to an abstraction:

Proposition 5.7. *For all $s \in L_{name}$: $s \sim_{name} \iota(N(s))$. If s is an L_{need}-WHNF and $N(s){\downarrow_{lazy}} v$ where v is an abstraction, then $s \sim_{need} \iota(v)$.*

Proof. We first show that for all expressions $s \in L_{name}$: $s \sim_{name} \iota(N(s))$. Since N is the identity mapping on `letrec`-free expressions of L_{name} and $N(s)$ is `letrec`-free, we have $N(\iota(N(s))) = N(s)$. Hence adequacy of N (Proposition 5.5) implies $s \sim_{name} \iota(N(s))$. Theorem 3.8 shows $N(s) \sim_{lazy} v$ and Proposition 5.5 show that $\iota(v) \sim_{name} \iota(N(s)) \sim_{name} s$. Finally, Theorem 4.10 shows the claim. ∎

Proposition 5.8. *For all $s, t \in L_{name}$: $s \leq_{name} t \implies N(s) \leq_{lazy} N(t)$.*

Proof. For this proof we treat L_{lazy} expressions as L_{name} expressions. Let $s, t \in L_{name}$ and $s \leq_{name} t$. By Proposition 5.7: $N(s) \sim_{name} s \leq_{name} t \sim_{name} N(t)$ and thus $N(s) \leq_{name} N(t)$. Since $N(s)$ and $N(t)$ are `letrec`-free, we can apply Lemma 5.6 and thus have $N(s) \leq_{lazy} N(t)$. ∎

Now we put all parts together, where $(N \circ W)(s)$ means $N(W(s))$:

Theorem 5.9. *N and $N \circ W$ are fully-abstract, i.e. for all L_{need}-expressions s, t: $s \leq_{need} t \iff N(W(s)) \leq_{lazy} N(W(t))$.*

6. On Simulation in L_{need}

First we show that finite simulation (see [SS08a]) is correct for L_{need}:

Proposition 6.1. *Let s, t be closed expressions in L_{need}. The following holds: $\big($For all closed abstractions r and for $r = \Omega$: $s\ r \leq_{need} t\ r\big) \iff s \leq_{need} t$.*

Proof. The $\Leftarrow$ direction is trivial. We show the nontrivial part. Assume that for all closed abstractions r and for $r = \Omega$: $s\ r \leq_{need} t\ r$. Then we transfer the problem to L_{lazy} as follows: $N(s)$ and $N(t)$ are closed expressions in L_{lazy}. Since the translation N is surjective, every closed L_{lazy}-expression is in the image of N. Thus for every closed L_{lazy}-expression r' that is an abstraction or Ω, there is some L_{need}-expression r, such that $N(r) = r'$. We have $N(s)\ r'{\downarrow} \implies N(t)\ r'{\downarrow}$, since $N(s\ r) = (N(s)\ N(r))$, and since N is fully abstract. We can apply Proposition 3.10 and obtain $N(s) \leq_{lazy} N(t)$. Now Theorem 5.9 shows $s \leq_{need} t$. ■

Now we show that the co-inductive definition of an applicative simulation results in a relation equivalent to contextual preorder. We show the following helpful lemma:

Lemma 6.2. *For all closed expressions s and r and L_{need}-WHNFs w: $(s\ r){\downarrow}w \iff \exists v : s{\downarrow}v \wedge (v\ r){\downarrow}w$.*

Proof. In order to prove "$\Rightarrow$" let $(s\ r){\downarrow}w$. There are two cases, which can be verified by induction on the length k of a reduction sequence $(s\ r) \xrightarrow{need,k} w$: $(s\ r) \xrightarrow{need,*} ((\lambda x.s')\ r) \xrightarrow{need,*} w$, where $s \xrightarrow{need,*} (\lambda x.s')$, and the claim holds. The other case is $(s\ r) \xrightarrow{need,*} (\texttt{letrec}\ Env\ \texttt{in}\ ((\lambda x.s')\ r)) \xrightarrow{need,*} w$, where $s \xrightarrow{need,*} (\texttt{letrec}\ Env\ \texttt{in}\ (\lambda x.s'))$. In this case $((\texttt{letrec}\ Env\ \texttt{in}\ (\lambda x.s'))\ r) \xrightarrow{need,(lapp)} (\texttt{letrec}\ Env\ \texttt{in}\ ((\lambda x.s')\ r)) \xrightarrow{need,*} w$, and thus the claim is proven. The "$\Leftarrow$"-direction can be proven in a similar way using induction on the length of reduction sequences. ■

Definition 6.3. We define in L_{need} a simulation $\leq_{b,need}$ as follows:
Let s, t be closed expressions and η be a binary relation on closed expressions. Then $s\ [\eta]_{need}\ t$ holds iff $s{\downarrow}_{need}v$ implies that $t{\downarrow}_{need}w$, and for all closed letrec-free abstractions r and for $r = \Omega$: $(v\ r)\ \eta\ (w\ r)$.

The relation $\leq_{b,need}$ is defined to be the greatest fixpoint of $[\cdot]_{need}$ within binary relations on closed expressions. Its open extension is denoted with $\leq^o_{b,need}$.

Proposition 6.4. *In L_{need}, for closed s, t the statement $s \leq_{b,need} t$ is equivalent to the following condition for s, t:*
$\forall n \geq 0$, and for all $r_i, i = 1, \ldots, n$ that may be closed letrec-free abstractions or Ω: $(s\ r_1 \ldots r_n){\downarrow}_{need} \implies (t\ r_1 \ldots r_n){\downarrow}_{need}$.

Proof. This follows from Lemma 6.2. The complete proof can be found in [SS10]. ■

Now we can prove that the simulation relation $\leq_{b,need}$ is equivalent to the contextual preorder on closed expressions:

Theorem 6.5. *For closed expressions s, t: $s \leq_{b,need} t \iff s \leq_{need} t$.*

Proof. Let $\leq_{need,0}$ the restriction of $\leq_{need}$ to closed expressions. It is easy to verify that $\leq_{need,0} \subseteq [\leq_{need,0}]_{need}$ and thus for closed expressions s, t: $s \leq_{need} t \implies s \leq_{b,need} t$. For the other direction let $s \leq_{b,need} t$. The criterion in Proposition 6.4 then implies that for all $n \geq 0 : s\ r_1\ \ldots\ r_n \downarrow_{need} \implies t\ r_1\ \ldots\ r_n \downarrow_{need}$, where r_i are closed letrec-free abstractions or Ω. Full-abstraction of $N \circ W$ (see Theorem 5.9) implies that $N(W(s\ r_1\ \ldots\ r_n)) \downarrow_{lazy} \implies N(W(t\ r_1\ \ldots\ r_n)) \downarrow_{lazy}$. Since N and W translate applications into applications, this also shows that $N(W(s))\ N(W(r_1))\ \ldots\ N(W(r_n)) \downarrow_{lazy} \implies N(W(t))\ N(W(r_1))\ \ldots\ N(W(r_n)) \downarrow_{lazy}$. Moreover, since every L_{lazy}-abstractions is an $N \circ W$-image of a letrec-free abstraction, we also conclude that $N(W(s)) \leq_{b,lazy,3} N(W(t))$. Now Theorem 3.7 and full abstraction of $N \circ W$ finally show $s \leq_{need} t$. ∎

Using the characterization in Proposition 6.4, it is possible to prove non-trivial equations, as shown in the example below.

Example 6.6. We consider two fixpoint combinators Y_1 and Y_2, where Y_1 is defined non-recursively, while Y_2 uses recursion. The definitions are: $Y_1 := \lambda f.((\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)))$, $Y_2 :=$ `letrec` $\mathit{fix} = \lambda f.f\ (\mathit{fix}\ f)$ `in` fix.

Using Proposition 6.4 we can easily derive that $Y_1\ K \sim_{need} Y_2\ K$ where $K := \lambda a.(\lambda b.a)$. This follows since $(Y_1\ K\ r_1 \ldots\ r_n)$ converges for all n. The obtained WHNF is equivalent (some `letrec`-bindings are garbage collected, and some variable-to-variable chains are eliminated) to (`letrec` $w = (x\ x), k = (\lambda a.(\lambda b.a)), x = (\lambda y.(k(yy)))$ `in` $\lambda u.w$). Normal-order reduction of $(Y_2\ K\ r_1 \ldots r_n)$ also always converges, where the WHNF is equivalent to the expression (`letrec` $w = (\mathit{fix}\ k), \mathit{fix} = (\lambda f.(f(\mathit{fix}\ f))), k = (\lambda a.(\lambda b.a))$ `in` $(\lambda u.w)$). Thus $Y_1\ K \sim_{need} Y_2\ K$ and both expressions are greatest elements w.r.t. $\leq_{need}$.

For open expressions, we can lift the properties from L_{lazy}, which also follows from full abstraction of $N \circ W$ and from Lemma 3.9.

Lemma 6.7. *Let s, t be any expressions, and let the free variables of s, t be in $\{x_1, \ldots, x_n\}$. Then $s \leq_{need} t \iff \lambda x_1, \ldots, x_n.s \leq_{need} \lambda x_1, \ldots, x_n.t$*

The results above imply the following theorem:

Main Theorem 6.8. $\leq_{need} = \leq^o_{b,need}$.

The main theorem implies that our embedding of the call-by-need letrec calculus into Abramsky's lazy lambda calculus is isomorphic w.r.t. the corresponding term models, i.e.:

Theorem 6.9. *The identical embedding $\iota : \mathcal{E}_{lazy} \to \mathcal{E}_{need}$ leads to an isomorphism between the term-models: Let the preorder, the quotients modulo $\sim_{lazy}$ and $\sim_{need}$, and the lifting of ι be marked with an overbar. Then $\overline{\iota} : \overline{\mathcal{E}_{lazy}} \to \overline{\mathcal{E}_{need}}$ is a bijection, and for all $s_1, s_2 \in \overline{\mathcal{E}_{lazy}}$: $s_1\ \overline{\leq_{lazy}}\ s_2 \iff \overline{\iota}(s_1)\ \overline{\leq_{need}}\ \overline{\iota}(s_2)$.*

7. The Call-by-Need Lambda Calculus of Ariola & Felleisen

For the sake of completeness we show that our results are transferable to the call-by-need lambda calculus with letrec of [Ari97]. The syntax is identical to the calculus L_{need}, but the standard reduction strategy of [Ari97] differs from our normal order reduction. In particular [Ari97] do not provide a standard reduction strategy but an equational system from which we will derive a standard reduction.

We will show that the normal order reduction and the standard reduction corresponding to the equational system of [Ari97] are interchangeable and thus define the same notion of contextual equivalence. As a further result we show that bisimilarity can also be based on the strategy according to [Ari97] and coincides with contextual equivalence.

We recall the standard reduction strategy of [Ari97]. We will denote the notions related to Ariola & Felleisen's calculus with a prefix or mark "AF", if necessary. First we introduce AF-evaluation contexts R_{AF} that play a role similar to our reduction contexts:

$$\begin{array}{lll} R_{AF} & ::= & [\cdot] \mid (R_{AF}\ s) \mid \texttt{letrec}\ Env\ \texttt{in}\ R_{AF} \mid \texttt{letrec}\ Env, x = R_{AF}\ \texttt{in}\ R_{AF}[x] \\ & & \mid \texttt{letrec}\ x_1 = R_{AF}, x_2 = R_{AF}[x_1], \ldots x_n = R_{AF}[x_{n-1}], Env\ \texttt{in}\ R_{AF}[x_n] \end{array}$$

In Figure 3 the standard reductions (abbreviated as AF-reduction) of [Ari97, Section 8] are shown where L is an $\mathcal{L}$-context as introduced in Sect. 3.2 and $R_{AF,i}, R'_{AF}, R''_{AF}$ are R_{AF}-contexts. The calculus of [Ari97] uses the notion of a black hole which represents a cyclic dependency of the form $\texttt{letrec}\ x_1 = R_{AF}[x_n], x_2 = R_{AF}[x_1], \ldots x_n = R_{AF}[x_1]$. In contrast to [Ari97], we do not consider a black hole to be an answer and therefore do not copy it in (deref) rules. This reflects the authors' intention, as shown by a similar copy restriction in [Ari94].

$$\begin{array}{ll}
(\beta_{need}) & R_{AF}[(\lambda x.s)\ r] \to R_{AF}[(\texttt{letrec}\ x = r\ \texttt{in}\ s)] \\
(\text{lift}) & R_{AF}[(\texttt{letrec}\ Env\ \texttt{in}\ L[\lambda x.s])\ r] \to R_{AF}[\texttt{letrec}\ Env\ \texttt{in}\ (L[\lambda x.s]\ r)] \\
(\text{deref}) & R_{AF,1}[\texttt{letrec}\ Env, x = \lambda y.s\ \texttt{in}\ R_{AF,2}[x]] \\
& \to R_{AF,1}[\texttt{letrec}\ Env, x = \lambda y.s\ \texttt{in}\ R_{AF,2}[\lambda y.s]] \\
(\text{deref}_{env}) & R'_{AF}[\texttt{letrec}\ x_1 = \lambda y.s, x_2 = R_{AF,2}[x_1], \ldots, x_n = R_{AF,n}[x_{n-1}], Env\ \texttt{in}\ R''_{AF}[x_n]] \\
& \to R'_{AF}[\texttt{letrec}\ x_1 = \lambda y.s, \\
& \qquad x_2 = R_{AF,2}[\lambda y.s], \ldots, x_n = R_{AF,n}[x_{n-1}], Env\ \texttt{in}\ R''_{AF}[x_n]] \\
(\text{assoc}) & R_{AF,1}[\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ L[\lambda x.s])\ \texttt{in}\ R_{AF,2}[x]] \\
& \to R_{AF,1}[\texttt{letrec}\ Env_1, Env_2, x = L[\lambda x.s]\ \texttt{in}\ R_{AF,2}[x]] \\
(\text{assoc}_{env}) & R'_{AF}[\texttt{letrec}\ x_1 = (\texttt{letrec}\ Env_2\ \texttt{in}\ L[\lambda x.s]), \\
& \qquad x_2 = R_{AF,2}[x_1], \ldots, x_n = R_{AF,n}[x_{n-1}], Env_1\ \texttt{in}\ R''_{AF}[x_n]] \\
& \to R'_{AF}[\texttt{letrec}\ Env_2, x_1 = L[\lambda x.s], \\
& \qquad x_2 = R_{AF,2}[x_1], \ldots, x_n = R_{AF,n}[x_{n-1}], Env_1\ \texttt{in}\ R''_{AF}[x_n]]
\end{array}$$

Figure 3: Reduction rules defining $\xrightarrow{AF}$

AF-answers are terms of the form $L[\lambda x.s]$. We write $s \xrightarrow{AF} t$, iff s is transformed into t by one of the rules in Fig. 3. If $s \xrightarrow{AF,*} v$ where v an AF-answer, then we write $s \downarrow_{AF} v$ or $s \downarrow_{AF}$, resp. if the answer v is not of interest. For the corresponding contextual approximation and equivalence we use $\leq_{AF}$ and $\sim_{AF}$ as symbols.

Compared to the reduction strategy in L_{need}, the AF-reduction performs the let-shiftings (lapp), (llet-in), (llet-e) as late as possible. A difference from L_{need} is that

sometimes reduction steps must be performed in deeply nested lets. For instance, in $\texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda z.z\ \texttt{in}\ (\lambda u.z)(\lambda uu))\ \texttt{in}\ x$ the L_{need} reduction will apply (llet-e) immediately, whereas AF will reduce $(\lambda u.z)(\lambda uu)$ first, and only then apply (assoc).

In [SS10] we prove:

Theorem 7.1. $\downarrow_{need} = \downarrow_{AF}$, $\leq_{need} = \leq_{AF}$ *and* $\sim_{need} = \sim_{AF}$.

Definition 7.2 (AF-simulation)**.** Let s, t be closed expressions and η be a binary relation on closed expressions. Then $s\ [\eta]_{AF}\ t$ holds iff $s{\downarrow_{AF}}v$ implies that $t{\downarrow_{AF}}w$, where v and w are answers, and for all closed letrec-free abstractions r and for $r = \Omega$: $(v\ r)\ \eta\ (w\ r)$. The relation $\leq_{b,AF}$ is defined to be the greatest fixpoint of $[\cdot]_{AF}$ within the binary relations on closed expressions. Its open extension is denoted with $\leq_{b,AF}^{o}$.

It remains to show that $\leq_{b,AF}^{o} = \leq_{AF}$. As a first step we derive an alternative characterization of $\leq_{b,AF}$. The proof can be found in [SS10].

Proposition 7.3. *For closed $s, t \in L_{need}$ the relation $s \leq_{b,AF} t$ is equivalent to: $\forall n \geq 0$, and for all $r_i, i = 1, \ldots, n$ that may be letrec-free abstractions or Ω: $(s\ r_1 \ldots r_n){\downarrow_{AF}} \implies (t\ r_1 \ldots r_n){\downarrow_{AF}}$.*

Proposition 7.4. $\leq_{b,need} = \leq_{b,AF}$

Proof. Since $\downarrow_{need} = \downarrow_{AF}$ the previous proposition and Proposition 6.4 show the claim. ∎

From Theorem 6.5 we already know that $\leq_{b,need}$ is equivalent to $\leq_{need}$ on closed expressions. Thus $\leq_{b,AF}$ is identical to $\leq_{need}$ on closed expressions. This easily extends to the open extension of $\leq_{b,AF}$. Thus we have:

Theorem 7.5. $\leq_{AF}\ = \leq_{b,AF}^{o}$

8. Conclusion

In this paper we show that co-inductive bisimulation, in the style of Howe, is equivalent to contextual equivalence in a deterministic call-by-need calculus with letrec (i.e. let with cyclic bindings). As a further work one may extend the proof to a call-by-need letrec calculus with case, constructors, and `seq`, but not to non-determinism, since counterexamples exist that show that contextual equivalence cannot be characterized by the usual notion of bisimulation.

Acknowledgement

The authors thank the anonymous reviewers for their valuable comments.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. A. Turner (ed.), *Research Topics in Functional Programming*, pp. 65–116. Addison-Wesley, 1990.

[Abr93] S. Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

[Ari94] Z. M. Ariola and J. W. Klop. Cyclic Lambda Graph Rewriting. In *Proc. IEEE LICS*, pp. 416–425. IEEE Press, 1994.

[Ari95] 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, San Francisco, California, 1995.

[Ari97] Z. M. Ariola and M Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997.

[Ari02] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117:95–168, 2002.

[Bar84] H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

[Fel91] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, 1991.

[Gol05] M. Goldberg. A variadic extension of Curry's fixed-point combinator. *Higher-Order and Symbolic Computation*, 18(3-4):371–388, 2005.

[How89] D. Howe. Equality in lazy computation systems. In *Proc. IEEE LICS*, pp. 198–203. 1989.

[How96] D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.

[Jef94] A. Jeffrey. A fully abstract semantics for concurrent graph reduction. In *Proc. IEEE LICS*, pp. 82–91. 1994.

[Man10] M. Mann and M. Schmidt-Schauß. Similarity implies equivalence in a class of non-deterministic call-by-need lambda calculi. *Information and Computation*, 208(3):276 – 291, 2010.

[Mar98] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *J. Funct. Programming*, 8:275–317, 1998.

[SS07] M. Schmidt-Schauß. Correctness of copy in calculi with letrec. In *Term Rewriting and Applications (RTA-18)*, *LNCS*, vol. 4533, pp. 329–343. Springer, 2007.

[SS08a] 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. of RTA 2008*, no. 5117 in LNCS, pp. 321–335. Springer-Verlag, 2008.

[SS08b] M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP TCS 2008*, *IFIP*, vol. 273, pp. 521–535. Springer, 2008.

[SS09a] M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Counterexamples to simulation in non-deterministic call-by-need lambda-calculi with letrec. Frank report 38, Inst. f. Informatik, Goethe-University, Frankfurt, 2009.

[SS09b] M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. Frank report 33, Inst. f. Informatik, Goethe-University, Frankfurt, 2009.

[SS10] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. Frank report 40, Inst. f. Informatik, Goethe-University, Frankfurt, 2010.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 311-324
http://rewriting.loria.fr/rta/

WEAK CONVERGENCE AND UNIFORM NORMALIZATION IN INFINITARY REWRITING

JAKOB GRUE SIMONSEN

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
E-mail address: simonsen@diku.dk

ABSTRACT. We study infinitary term rewriting systems containing finitely many rules. For these, we show that if a weakly convergent reduction is not strongly convergent, it contains a term that reduces to itself in one step (but the step itself need not be part of the reduction). Using this result, we prove the starkly surprising result that for any orthogonal system with finitely many rules, the system is weakly normalizing under weak convergence iff it is strongly normalizing under weak convergence iff it is weakly normalizing under strong convergence iff it is strongly normalizing under strong convergence.

As further corollaries, we derive a number of new results for weakly convergent rewriting: Systems with finitely many rules enjoy unique normal forms, and acyclic orthogonal systems are confluent. Our results suggest that it may be possible to recover some of the positive results for strongly convergent rewriting in the setting of weak convergence, if systems with finitely many rules are considered. Finally, we give a number of counterexamples showing failure of most of the results when infinite sets of rules are allowed.

1. Introduction

In term rewriting, weak normalization is the property that every term has a normal form ("there exists at least one reduction to normal form"), whereas strong normalization, also called termination, is the property that every reduction from every term is finite ("all reductions will eventually lead to a normal form"). For some subclasses of term rewriting systems (TRSs), it is known that the property of *uniform normalization* holds: A system is weakly normalizing iff it is strongly normalizing. This property holds, for example, for the class of *orthogonal, non-erasing* systems, that is, every variable occurring in the left-hand side of a rule must also occur on the right-hand side of that rule [18].

In the elegant paper [19], Klop and de Vrijer argue that when lifting the concepts of weak and strong normalization to infinitary rewriting, only weak normalization should be lifted in the obvious way: From every term, there is a, possibly infinite, reduction to normal form. But strong normalization should be treated differently. In the infinitary setting, strong normalization should instead be the property that every well-behaved reduction is *convergent*: Every possible infinite reduction satisfying some very basic integrity constraints

1998 ACM Subject Classification: F.4.1, F.4.2.

Key words and phrases: Infinitary rewriting, weak convergence, uniform normalization.

© J.G. Simonsen
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.311

should have a well-defined limit to which it converges, regardless of whether that limit is a normal form.

Klop and de Vrijer prove the following remarkable result:

Theorem 1.1 (Klop, de Vrijer [19]). *For any orthogonal iTRS R: All strongly continuous reductions can be extended to strongly convergent reductions ("R is strongly normalizing") iff every term of R reduces to a normal form by a strongly convergent reduction ("R is weakly normalizing").*

Thus, the theorem states that strong and weak normalization *coincide* in the setting of strongly convergent reductions.

Strong convergence means that not only do reductions converge in the complete metric space of (potentially infinite) trees [1, 3] (called *weak* convergence), but the number of rewrite steps occurring at each finite depth is finite along *any* reduction. While weak convergence was the first notion of infinitary rewriting studied [4], it has by now been established that weak convergence in general does not have the desirable properties normally true for syntactically well-behaved (that is, *orthogonal*) rewriting systems[25] while strong convergence does [11, 9].

As a contribution towards showing positive results for weak convergence, we will show that the equivalence in Theorem 1.1 may be extended to hold in the setting of weakly convergent rewriting with the proviso that we only consider systems with a finite number of rules. This should be contrasted with the counterexamples given in [25] that relied crucially on systems with an *infinite* number of rules.

The key to our results is a characterization of the difference between weakly and strongly convergent rewriting: If there exists a weakly convergent reduction $s \to t_1 \to t_2 \to \cdots t$ that is not also strongly convergent, there will be some—possibly infinite—term t' occurring in $s \to t_1 \to t_2 \to \cdots t$ such that $t' \to t'$. That is, the rewrite system *admits a cycle of length* 1.

The observation that the existence of cycles of lengths 1 is *the* crucial difference between strong and weak convergence is the focal point of the paper and will furnish a small host of derivative results. The core results of the paper are Theorems 3.16 and 4.1; these contain the main (and somewhat surprising) equivalence results, in particular the latter contains the result that strong and weak normalization in weak and strong convergence *all* coincide for systems with a finite number of rules.

Another interesting new result derived from the above concerns confluence in weakly convergent rewriting: If an orthogonal system with a finite number of rules is acyclic, it will also be confluent. We believe this to be one of the first general and non-trivial confluence results in weakly convergent rewriting.

A remark on concessions to readability

The author has made a concession to readability in this paper, at the expense of further generality: All of the results are proven for (first-order) infinitary term rewriting systems (iTRSs). Bar the explicit counterexamples we give, none of our proofs rely on the first-order nature of these systems. The results of this paper *also* hold true for both infinitary lambda calculus [10] and for infinitary combinatory reduction systems (iCRSs) [14, 15], as the reasoning employed in our proofs essentially only uses the machinery of metric spaces and the abstract notion of depth in the distinction between weak and strong convergence.

The author believes that an abstract account of all our results can be given in the topological setting of Kahrs' work on meta-theory for infinitary rewriting [6], but we feel that a very concrete "syntactical" account such as we give may reach a wider audience.

1.1. Related work

Uniform normalization has been studied in various forms of finitary rewriting, including first-order term rewriting [23, 17], lambda calculus [26, 22] and higher-order rewriting [16].

The original study on weakly convergent rewriting [4] provided a number of results related to normalization. Many of these results concern *top-termination*, a concept equivalent to strong normalization in the setting of strongly convergent reductions, and most of the results thus also hold for the latter, but have been subsumed by later work in that setting. In addition, the definition of "normal form" in [4] concerns infinite terms s with the property that the only possible reduct of s is s. This is in contrast to this and most other papers where infinite normal forms are terms that allow no rewrite steps starting from them.

Klop and de Vrijer were the first to extend the study of uniform normalization to the infinitary setting [19]; they prove Theorem 1.1 and several other interesting results, but do not consider weak convergence.

Ketema extends Theorem 1.1 and several other results of [19] to higher-order infinitary rewriting in the setting of strong convergence [13].

Lucas proves several results related to normalization and confluence in weakly convergent rewriting [21]; all those results concern constructor systems.

Zantema considers strong normalization in a general setting allowing infinite left-hand sides in rules [27]. He characterizes the property of every reduction being strongly convergent using a novel variation of monotone algebras and shows applicability of a simple generalization of termination-by-matrix-analysis from the finitary setting to the setting of strongly convergent infinitary rewriting. It has later been pointed out that Zantema's results hold for reductions of length $\leq \omega$, and that continuous algebras are needed for reductions of greater ordinal length [5]. Weak convergence is not considered in these papers.

Kahrs has taken the first steps in establishing a working theory of general infinitary rewriting that in particular tackles the inherent difficulties of weakly convergent rewriting [6, 7]. He considers modularity of strong normalization in [7], but not the equivalence of strong and weak convergence.

Rodenburg defines infinite reduction sequences over terms with symbols having infinite arities, but his terms are built by induction over such function symbols, hence have only finite depth [24]. It is not hard to see that every infinitary term rewriting system in the sense of the present paper (finite arities, but a liberal notion of convergence) can be simulated by one of Rodenburg's systems if the latter is allowed to have an infinite number of function symbols and rules. Unfortunately, this is the case even if the original system has only a finite number of symbols and rules, the most pertinent restriction in the present paper. Furthermore, even though Rodenburg's reductions correspond to weakly convergent reductions, his main results concern termination and a generalization of Newman's lemma for systems with a complexity measure satisfying certain properties (socalled *weakly* and *strongly descending* systems). These properties do not seem to be satisfied by a sufficiently large class of systems, and we hence feel that a technical exposition involving translation to his systems would not be fruitful in our setting.

Finally, the author of the present paper shows that most of the usual methods for procuring nice-to-have properties of orthogonal systems do not hold in the setting of weak convergence [25]. In particular, confluence of (even non-collapsing) orthogonal systems does not hold when an infinite number of rules is allowed.

2. Preliminaries on infinitary term rewriting

Throughout the paper, we presuppose a working knowledge of ordinals [20]; the least infinite ordinal is denoted by ω, the least uncountable ordinal by Ω. The general theory for infinitary rewriting is laid out for weak convergence in [4] and for strong convergence in [9]. We give the briefest of definitions below to keep the paper self-contained.

Definition 2.1. Assume a denumerably infinite set V of variables and a first-order signature Σ of function symbols. The set of (finite) *terms* over Σ with variable set V, denoted $Ter(\Sigma, V)$, is defined inductively as follows: (i) every $x \in V$ is a term, (ii) if $f \in \Sigma$ is an n-ary function symbol and $s_1, \ldots, s_n$ are terms, then $f(s_1, \ldots, s_n)$ are terms. A *position* is any finite sequence of positive integers. Given a term s, the set of *positions of* s is the least set of positions such that (i) the empty string ϵ is a position, and (ii) if $s = f(t_1, \ldots, t_i, \ldots, t_m)$ and p is a position of term t_i, then $i \cdot p$ is a position of s and we say that t_i is *the subterm of* s *at position* $i \cdot p$, writing $s|_{i \cdot p} = t_i$. The *length* of a position is denoted $|p|$. The *depth* of a finite term s is the length of the longest position in s.

Definition 2.2. The *term metric* is the metric on $Ter(\Sigma, V)$ defined by $d(s,t) = 0$ if $s = t$ and if $s \neq t$ by $d(s,t) = 2^{-k}$ where k is the length of the shortest position at which s and t differ. The set of *finite and infinite terms*, denoted $Ter^\infty(\Sigma, V)$ is the metric completion of $Ter(\Sigma, V)$ with respect to the metric d—that is, the set obtained by augmenting $Ter(\Sigma, V)$ by the set of all limits of Cauchy sequences of elements of $Ter(\Sigma, V)$. An *infinitary term rewriting system* (iTRS) (over $Ter^\infty(\Sigma, V)$) is a set R of pairs (l, r), written $l \to r$ where (i) $l \in Ter(\Sigma, R) \setminus V$, (ii) $r \in Ter^\infty(\Sigma, V)$, and (iii) all variables in r occur in l. The pairs $l \to r$ are called *rules*. A rule is *collapsing* if $r \in V$.

Thus, if $\Sigma = \{f, g\}$ where f and g a binary and unary symbols respectively, then $s = f(g(x), s) = f(g(x), f(g(x), f(g(x), \ldots)$ is an infinite term (it is the limit of the Cauchy sequence $f(g(x), y), f(g(x), f(g(x), y)), f(g(x), f(g(x), f(g(x), y))), \ldots$.

Observe that left-hand sides of rules are finite, but right-hand sides may be infinite. This is a standard technical convenience that also serves to retain decidability of applicability of a given rewrite rule at a given position in a term[1].

Definition 2.3. A *substitution* is a map $\theta : V \longrightarrow Ter^\infty(\Sigma, V)$ (where θ is usually specified only on a finite subset of V). Any substitution can be extended to a map $\theta : Ter^\infty(\Sigma, V) \longrightarrow Ter^\infty(\Sigma, V)$ by setting $\theta(f(s_1, \ldots, s_n)) = f(\theta(s_1), \ldots, \theta(s_n))$ for all n-ary function symbols $f \in \Sigma$. A *one-hole context* is a term over $Ter^\infty(\Sigma, V \cup \Box)$ with exactly one occurrence of $\Box$. Let t be any term, and σ be the substitution $\{\Box \mapsto t\}$; we write $s = C[t]$ if $s = \sigma(C[])$ A *rewrite step* of rule $l \to r$ is a pair $C[\theta(l)] \to C[\theta(r)]$ where θ is a substitution. The rewrite step *occurs at position* p (and *at depth* $|p|$) if p is the position of $\Box$ in $C[]$.

Thus, if $R = \{f(x) \to g(x, a)\}$, then $f(f(a)) \to f(g(a, a))$ is a rewrite step at position 1 (and at depth 1).

[1]Note that left-linearity is also required for decidability for iTRSs. Both left-linearity and fully-extendedness are required for decidability in iCRSs.

Definition 2.4. An iTRS R is said to be left-linear if, for all of its rules $l \to r$, every variable x occurs at most once in l. R is said to be *orthogonal* if it is left-linear, and for all pairs of rules $(l_1 \to r_1, l_2 \to r_2)$, the following holds: If there is a context $C[]$ with the hole at position p and substitutions σ, θ such that $\sigma(l_1) = C[\theta(l_2)]$, then either (i) a variable in l_1 is at a prefix position of p, or (ii) $p = \epsilon$ and $l_1 \to r_1 = l_2 \to r_2$.

Example 2.5. The iTRSs $R_1 = \{f(x,x) \to x\}$ and $R_2 = \{f(g(x),a) \to a, g(a) \to a\}$ are not orthogonal. The iTRS $R_3 = \{f(g(x),a) \to a, f(a,x) \to g(x)\}$ is orthogonal.

When $C[]$ is a one-hole context, we usually write C^ω for the infinite term $C[C[C[\cdots]]]$.

Definition 2.6. Let α be an ordinal. A transfinite reduction with domain $\alpha > 0$ is a sequence of (terms,positions,rules) $(s_\beta, p_\beta, (l \to r)_\beta)_{\beta<\alpha}$ such that, for each $\beta + 1 < \alpha$ we have $s_\beta \to s_{\beta+1}$ by contraction of a redex of rule $l \to r$ at position p_β. The reduction is *open* if α is a limit ordinal and *closed* if α is a successor ordinal. The *length* of an open reduction is α, and the length of a closed reduction $\alpha - 1$.

A transfinite reduction is *weakly* (also known as *Cauchy*) continuous if for every limit ordinal $\gamma < \alpha$, it holds that s_β converges to s_γ in the metric d as β approaches γ from below. The reduction is *weakly convergent* if it is weakly continuous and closed. We write $s \twoheadrightarrow_{\mathrm{W}} t$ if there is a weakly convergent reduction $(s_\beta)_{\beta<\alpha'+1}$ with $s_0 = s$ and $t = s_{\alpha'}$.

For every rewrite step $s_\beta \to s_{\beta+1}$, let d_β denote the depth of the contracted redex. The reduction is strongly continuous if it is weakly continuous and if, for every limit ordinal $\gamma < \alpha$, the depth d_β tends to infinity as β approaches γ from below. The reduction is strongly convergent if it is strongly continuous and closed. We write $s \twoheadrightarrow_{\mathrm{S}} t$ if there is a strongly convergent reduction $(s_\beta)_{\beta<\alpha'+1}$ with $s = s_0$ and $t = s_{\alpha'}$.

The requirement that the length, α, of a convergent reduction be a successor ordinal is to ensure that the limit term of a continuous reduction is included. Most of the literature on infinitary rewriting has focused on strong convergence as it is the more pliable of the two notions of convergence under technical manipulation, and as the demand that rewrite steps eventually occur deeper and deeper corresponds to computational intuition about manipulation of potentially infinite data structures in finite time.

Definition 2.7. An *extension* of a reduction $S : s_0 \to s_1 \to \cdots$ of length α is a reduction $T : t_0 \to t_1 \to \cdots$ of length $\zeta \geq \alpha$ such that for all $0 \leq \beta + 1 < \alpha$, $s_\beta = t_\beta$ and the steps $s_\beta \to s_{\beta+1}$ and $t_\beta \to t_{\beta+1}$ are contractions of redexes at identical positions and of identical rules (informally: S is a prefix of T).

Example 2.8. Consider the orthogonal iTRS $\{a(x) \to a(b(x))\}$. The reduction

$$a(x) \to a(b(x)) \to a(b(b(x))) \to \cdots a(b^\omega)$$

is weakly convergent, but not strongly convergent. Observe that there are no strongly convergent reductions from $a(x)$ to $a(b^\omega)$.

The reduction

$$a^\omega \to a(b(a^\omega)) \to a(b(a(b(a^\omega)))) \to \cdots a(b(a(b(a(b(\cdots))))))$$

is strongly convergent, hence also weakly convergent.

Strongly convergent reductions are of countable length:

Proposition 2.9. *Every strongly convergent reduction is of countable length.*

Proof. See for example [11, Lemma 3.5]. ■

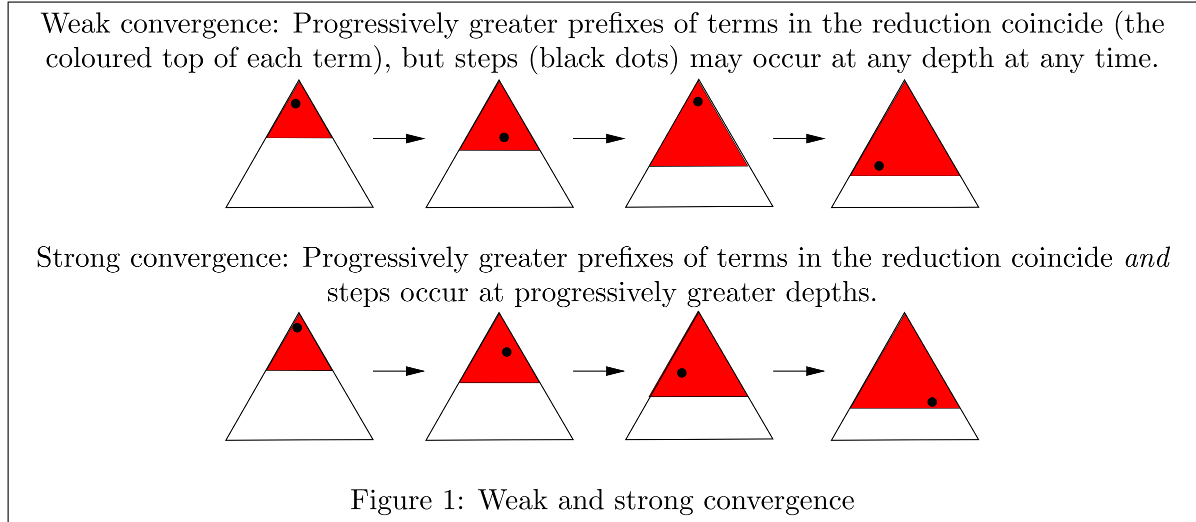


Figure 1: Weak and strong convergence

2.1. Preliminaries on weak and strong normalization

The concept of normal form is lifted to infinitary rewriting in the obvious way:

Definition 2.10. A *normal form* is a term t such that no rewrite step starts from t.

A term s is *normalizing under weak convergence*, denoted $WN^{\infty}_{\mathrm{W}}(s)$, if there is a normal form t such that $s \twoheadrightarrow_{\mathrm{W}} t$. The term s is *normalizing under strong convergence*, denoted $WN^{\infty}(s)$ if there is a normal form t such that $s \twoheadrightarrow_{\mathrm{S}} t$.

The iTRS R is normalizing under weak convergence, denoted $WN^{\infty}_{\mathrm{W}}(R)$, if every s satisfies $WN^{\infty}_{\mathrm{W}}(s)$, and R is normalizing under strong convergence, denoted $WN^{\infty}(R)$, if every term s satisfies $WN^{\infty}(s)$.

It is by now well-established [8, 19, 27, 13] that the proper way to sensibly extend the notion of *strong* normalization ("termination") to infinitary rewriting is to require that every strongly continuous reduction can be extended to a strongly convergent one (informally: "If the reduction has well-defined prefixes, then it is convergent"). We follow Ketema [13] in the wording of the definition below, extending it to strong normalization under *weak* convergence in the obvious way.

In the setting of *weak* convergence, we might define strong normalization analogously, but this would be a poor choice of nomenclature—it could be that no *normalization* occurs. For example, using the rule $a \to a$, the weakly continuous reduction $a \to a \to a \to \cdots$ of length ω can be extended to a weakly convergent reduction by simply adding a at the end; but a has no normal form. So, certainly, the ability to extend any weakly continuous reduction to a weakly convergent one does *not* imply that a has a normal form. We therefore prefer to use the term *extendable* for this property of weakly convergent reductions.

One could argue that there is need for a generalization of the concept of termination for weakly convergent rewriting; by analogue with termination in finitary rewriting, we have chosen what we believe to be the most obvious generalization: To extend the definition of $EXT^{\infty}_{\mathrm{W}}(R)$ with the demand that there is a "maximal" length of reductions. As we shall later see the existence of such an ordinal is *equivalent* to the assumption that every weakly convergent reduction is also strongly convergent (as always in this paper, when the system has a finite number of rules).

(Finitary) term rewriting	Infinitary rewriting (strong convergence)	Infinitary rewriting (weak convergence)
(Weak) normalization (WN): Every term has a finite reduction to normal form	WN^∞: Every term has a strongly convergent reduction to normal form	WN^∞_{W}: Every term has a weakly convergent reduction to normal form
Termination (SN): Every reduction is finite	SN^∞: Every strongly continuous reduction can be extended to a strongly convergent reduction	SN^∞_{W}: Every weakly continuous reduction can be extended to a weakly convergent reduction *and* there is an ordinal upper bound on the length of weakly convergent reductions

Table 1: Corresponding notions for normalization in ordinary and infinitary rewriting

Definition 2.11. Term s is said to be *extendable under weak convergence*, denoted $EXT^\infty_{\mathrm{W}}(s)$ if, for every ordinal α, any open, weakly continuous reduction of length α starting from s can be extended to a weakly convergent reduction. A term s is *strongly normalizing under strong convergence*, denoted $SN^\infty(s)$ if, for every ordinal α, any open, strongly continuous reduction of length α can be extended to a strongly convergent reduction. The iTRS R is said to be extendable under weak convergence, denoted $EXT^\infty_{\mathrm{W}}(R)$,if all of its terms s satisfy $EXT^\infty_{\mathrm{W}}(s)$, and strongly normalizing under strong convergence, denoted $SN^\infty(R)$, if all of its terms s satisfy $SN^\infty(s)$. Finally, R is said to be *strongly normalizing under weak convergence*, denoted $SN^\infty_{\mathrm{W}}(R)$ if $EXT^\infty_{\mathrm{W}}(R)$ and there exists an ordinal α such that every weakly convergent reduction has length $< \alpha$.

Our definition of SN^∞_{W} is similar to Rodenburg's [24]: Namely that each term t has an upper bound on the length of convergent reductions starting from t (note that Rodenburg's notion of term and convergence differs from the one in modern infinitary rewriting). For iTRSs, the two notions are equivalent: If $SN^\infty_{\mathrm{W}}(R)$, there is trivially an upper bound on the length of convergent reductions starting from any term. Conversely, let α_t be an upper bound on the length of convergent reductions starting from term t; as the set of all infinite terms can be indexed by the least uncountable ordinal, $\gamma = \sup_t \alpha_t$ is an ordinal and is an upper bound on the length of all convergent reductions, whence $SN^\infty_{\mathrm{W}}(R)$.

Example 2.12 (Ketema [13]). The iTRS $R_1 = \{a \to a\}$ satisfies $EXT^\infty_{\mathrm{W}}(R_1)$ as the only possible reductions are those starting $a \to a \to \cdots$, all of which are weakly convergent.

The iTRS $R_2 = \{f(x) \to g(f(x))\}$ satisfies both $EXT^\infty_{\mathrm{W}}(R_2)$ and $SN^\infty(R_2)$ as (i) exactly one new redex is created by contraction of a redex and, (ii) for each depth m, every term has only a finite number of redexes at depth at most m and the depth of the new redex created by a contraction is exactly one greater than that of the creating redex, (iii) the rewrite rule strictly increases the depth of its argument x.

The iTRS $R_3 = \{a \to b, b \to a\}$ satisfies neither $EXT^\infty_{\mathrm{W}}(R_3)$, nor $SN^\infty(R_3)$, as $a \to b \to a \to b \to \cdots$, and the only convergent reductions starting from a (or b) are finite.

We end this section by noting that in systems with a finite number of rules, normalization of individual terms does not depend on the notion of convergence used:

Proposition 2.13. *Let R be an orthogonal iTRS consisting of a finite number of rules and s be a term. If $s \twoheadrightarrow_{\mathcal{W}} t$ where t is a normal form, then $s \twoheadrightarrow_{\mathcal{S}} t$.*

Proof. See e.g. [11, Thm. 9.1]. ∎

3. Weak, but not strong, convergence entails existence of a cycle of length 1

This section is devoted to proving that every weakly convergent reduction $s \twoheadrightarrow_{\mathrm{W}} t$ that is not strongly convergent can be written as $s \twoheadrightarrow_{\mathrm{W}} t' \twoheadrightarrow_{\mathrm{W}} t$ where t' is a term that reduces to itself in one step: $t' \to t'$; i.e., there is a *cycle of length 1*.

Before outlining the proof idea, we introduce the concept of a *cofinal map*[2].

Definition 3.1. If $g : \beta \longrightarrow \alpha$, then g is *cofinal* if, for all $\gamma < \alpha$, there exists $\zeta < \beta$ such that $\gamma < g(\zeta)$.

Definition 3.2. Let $S : s \twoheadrightarrow_{\mathrm{W}} s_\alpha$ be a weakly convergent reduction of length α, let $\alpha' \leq \alpha$, and let $g : \beta \longrightarrow \alpha'$ be strictly monotonic. The *g-pick* of S is a sequence $(s_{g(\gamma)})_{\gamma \in \beta}$ where each $s_{g(\gamma)}$ occurs in S. We say that the g-pick $(s_{g(\gamma)})_{\gamma \in \beta}$ is *induced* by g. The g-pick is said to be *cofinal* if g is cofinal. We shall occasionally suppress the g and speak merely of a *pick*.

A pick is not a reduction, nor does it necessarily induce a reduction in any meaningful sense. Picks are simply a way of "picking out" terms from the reduction S. That the function g is cofinal means that the terms in the pick occur "densely" before the term $s_{\alpha'}$. For example, if $\alpha = \omega^2$ and $\alpha' = \omega$, the pick induced by $g : \omega \to \omega$ where $g(\gamma) = \gamma \cdot 2 + 1$ is cofinal.

The idea of the proof of the splitting $s \twoheadrightarrow_{\mathrm{W}} t' \twoheadrightarrow_{\mathrm{W}} t$ is quite simple: There is a position p such that only a finite number of steps occur above p in $s \twoheadrightarrow_{\mathrm{W}} t$ and such that an infinite number of steps occur at p; due to the fact that only a finite number of rules are present, one rule, $l \to r$, must be used an infinite number of times at p. By taking the least limit ordinal α' such that an infinite number of such steps occur before α', we employ the fact that the prefix, $s \twoheadrightarrow_{\mathrm{W}} t'$, of $s \twoheadrightarrow_{\mathrm{W}} t$ of length α' is weakly continuous to show that the sequence of subterms at position p converges at α', and that the subterm $t'|_p$ satisfies $t'|_p = \theta(l) = \theta(r)$ for some substitution θ. The result then follows immediately, as $t' = t'[\theta(l)]_p \to t'[\theta(r)]_p = t'[\theta(l)]_p = t'$.

Proposition 3.3. *Let R be an iTRS consisting of a finite number of rules, and let S : $s \twoheadrightarrow_{W} t$ be a weakly convergent reduction of length α that is not strongly convergent. Then there is a rule $l \to r \in R$, a limit ordinal $\alpha' \leq \alpha$, an ordinal $\gamma < \alpha'$, a position p and an infinite pick $(s_{g(\zeta)})_{\zeta<\beta}$ induced by a cofinal $g : \beta \to \alpha'$ such that:*

- *For all $\gamma < \delta < \alpha'$, no step $s_\delta \to s_{\delta+1}$ occurs at a position $q < p$ in S.*
- *For every $\zeta < \beta$, the rewrite step $s_{g(\zeta)} \to s_{g(\zeta)+1}$ occurs at position p and employs rule $l \to r$.*

Proof. As any finite reduction is strongly convergent, we have $\alpha \geq \omega$. By standard results (see for example [9, Ex. 12.3.6]), there is a position of minimal depth m in S such that the number of redex contractions at depth m is infinite, as S would otherwise be strongly convergent. Observe that as m is minimal among such depths, there is an ordinal $\gamma < \alpha$ such that for any $\gamma \leq \delta < \alpha$, the rewrite step $s_\delta \to s_{\delta+1}$ does not occur at a position of length $< m$.

Let $\alpha' \leq \alpha$ be the least limit ordinal such that an infinite number of contractions at depth m occur in the prefix of $s \twoheadrightarrow_{\mathrm{W}} t$ of length α'. By weak convergence of S, this prefix converges to some term t'.

[2]Nomenclature taken from [7]. Cofinality is a standard notion in set theory, see e.g. [20, Ch. 1]; a related concept from [25] is that of an α'-*frequent* property.

Claim: There is an infinite, cofinal pick induced by a $g : \beta \to \alpha'$ such that the redex employed in the step $s_{g(\zeta)} \to s_{g(\zeta)+1}$ occurs at depth m for every $\zeta < \beta$.

The claim follows by contradiction: If the claim did not hold, there would be a limit ordinal $\alpha'' < \alpha'$ with an infinite number of contractions at a depth $\leq m$, contradicting the above observations.

As t' has only a finite number of positions at depth m, there must be a position p with $|p| = m$ and an infinite cofinal pick induced by $g' : \beta' \to \alpha'$ in which every rewrite step occurs at p.

As R consists of a finite number of rules and the pick induced by g' is infinite, the pigeon-hole principle yields that there is a rule $l \to r$ and an infinite pick induced by $g'' : \beta'' \to \alpha'$ with $\beta'' \leq \beta'$ in which every rewrite step occurs at p and is a contraction of a redex of rule $l \to r$. As α' was the minimal limit ordinal with an infinite number of steps at depth $m = |p|$, the pick induced by g must be cofinal, as desired. ∎

Example 3.4. To illustrate the proof of Proposition 3.3, we give a short example.

Let R be the (non-orthogonal) iTRS consisting of the rules $\{f(x) \to f(g(x)), g(x) \to g(f(x))\}$, and consider the following reduction S where we have underlined the root symbol of the contracted redex in each step:

$$\begin{array}{rcl} \underline{g}(x) & \to & g(\underline{f}(x)) \\ & \to & g(f(\underline{g}(x))) \\ & \to & g(\underline{f}(g(f(x)))) \\ & \to & g(f(g(\underline{g}(f(x))))) \\ & \to & g(\underline{f}(g(g(g(f(x)))))) \\ & \to & \cdots \\ & \cdots & g(f(g^\omega)) \end{array}$$

where we assume that ω steps are performed. The above reduction is weakly convergent, but not strongly so. Writing the reduction as $g(x) = s_0 \to s_1 \to s_2 \to \cdots s_\omega = g(f(g^\omega))$, the step $s_0 \to s_1$ occurs at position ϵ; for $k \geq 1$, the step $s_{2k} \to s_{2k+1}$ occurs at position 1, and the step $s_{2k+1} \to s_{2k+2}$ occurs at position $1 \cdot 1^k$. Define $g : \omega \to \omega$ by $g(k) = 2k$. Then $(s_{2k})_{k \in \omega}$ is a cofinal pick induced by g where each step occurs at position $p = 1$, and each step is of the rule $f(x) \to f(g(x))$; note that there are only a finite number of steps occurring above p in S.

We shall also need the standard concept of a unifier:

Definition 3.5. Let s and t be terms. A *unifier* of s and t is a substitution θ such that $\theta(s) = \theta(t)$.

We refer to [2] for details on unification of infinite terms.

Proposition 3.6. *If $l \to r$ is a rule and θ is a unifier of l and r, then $\theta(l) \to \theta(l)$*

Proof. By definition of the rewrite relation, we have $\theta(l) \to \theta(r)$. As θ is a unifier of l and r, we have $\theta(r) = \theta(l)$, and the result follows. ∎

Lemma 3.7. *Let R be a an iTRS with a finite number of rules. With notation as in Proposition 3.3, let $s = s_0 \to s_1 \to s_2 \to \cdots s_\xi \to s_{\xi+1} \to \cdots s_{\alpha'} = t'$ be the closed prefix of $s \twoheadrightarrow_w t$ of length α'. Then there is a rule $l \to r \in R$ and a (countable!) sequence $(\sigma_\xi)_{\xi<\omega}$ of substitutions such that:*

- *For every x occurring in l, the sequence $(\sigma_\xi(x))_{\xi<\omega}$ converges in the tree metric to some term t_x.*
- *The substitution σ_ω defined by $\sigma_\omega(x) = t_x$ is a unifier of l and r.*
- $t'|_p = \sigma_\omega(l) = \sigma_\omega(r)$.

Proof. As $s \twoheadrightarrow_{\mathrm{W}} t$ is weakly convergent, it converges at α' to t'. As the g-pick $(s_{g(\gamma)})_{\gamma\in\beta}$ is cofinal and $g : \beta \to \alpha'$, the sequence $(s_{g(\gamma)})_{\gamma<\beta}$ converges to t'. For each $\gamma < \beta$, we have $s_{g(\gamma)}|_p = \sigma_\gamma(l)$ for some substitution σ_γ.

By weak convergence, there is thus for each natural number m, a substitution σ_m such that $d(\sigma_m(l), \sigma_m(r)) < 2^{-m}$ and $d(\sigma_m(l), t'|_p) < 2^{-m}$.

Let p_i be the position of any occurrence of variable x_i in l. By the above observations, $k \geq m$ implies that $d(\sigma_m(l)|_{p_i}, \sigma_k(l)|_{p_i}) < 2^{-m+|p_i|}$. But $\sigma_k(l)|_{p_i} = \sigma_k(x_i)$, whence the sequence $(\sigma_k(x_i))_{k<\omega}$ converges in the tree metric to some term t_i. To see that σ_ω is a unifier of l and r, consider $d(\sigma_\omega(l), \sigma_\omega(r))$. For each natural number m we then have:

$$\begin{array}{rcl} d(\sigma_\omega(l), \sigma_\omega(r)) & \leq & d(\sigma_\omega(l), \sigma_{m+2}(l)) + d(\sigma_{m+2}(l), \sigma_\omega(r)) \\ & \leq & 2^{-(m+2)} + d(\sigma_{m+2}(l), \sigma_{m+2}(r)) + d(\sigma_{m+2}(r), \sigma_\omega(r)) \\ & \leq & 2^{-(m+2)} + 2^{-(m+2)} + 2^{-(m+2)} \\ & < & 2^{-m} \end{array}$$

As m was arbitrary, we obtain $\sigma_\omega(l) = \sigma_\omega(r)$, and as the sequence $(\sigma_k(l))_{k<\omega}$ converges to $t'|_p$ in the tree metric, we have $t'|_p = \sigma_\omega(l) = \sigma_\omega(r)$. ∎

We have come to the main ancillary result of the paper:

Theorem 3.8. *Let R consist of a finite number of rules. If there exists a weakly convergent reduction $s \twoheadrightarrow_W t$ that is not strongly convergent, then $s \twoheadrightarrow_W t$ may be written as $s \twoheadrightarrow_W t' \twoheadrightarrow_W t$ where t' is a term with $t' \to t'$.*

Proof. Let notation be as in Proposition 3.3 and write $s \twoheadrightarrow_{\mathrm{W}} t$ as $s \twoheadrightarrow_{\mathrm{W}} t' \twoheadrightarrow_{\mathrm{W}} t$ where by Lemma 3.7 we have $t'|_p = \sigma_\omega(l) = \sigma_\omega(r)$ for some substitution σ_ω. By Proposition 3.6, we then have $t'|_p \to t'|_p$ and thus $t' \to t'$. ∎

Theorem 3.8 fails in the presence of an infinite number of rules:

Example 3.9. Let R be the orthogonal iTRS with infinite rule set $\{g^n(c) \to g^{n+1}(c) : n \geq 1\}$. Then there is a weakly convergent reduction $g(c) \to g(g(c)) \to \cdots g^\omega$ where the redex is contracted at the root in each step, whence the reduction is not strongly convergent. Observe that no term on the form $g^n(c)$ is cyclic (in fact, R is acyclic), and g^ω is a normal form; hence, the assumption of finiteness of the set of rules in Theorem 3.8 cannot be omitted.

Remark 3.10. An anonymous referee has kindly directed the author's attention to [12] where Lemma 4.3.2 states (using the terminology of the present paper) that for an orthogonal iTRS, there does *not* exist a rule $l \to r$ where l unifies with r iff iff all weakly convergent reductions are strongly convergent iff all weakly convergent reductions are top-terminating. Whereas Lemma 4.3.2 is formulated for arbitrary iTRSs, the authors of [12] almost certainly meant for their iTRSs to have a finite number of rules: Example 3.9 exhibits an orthogonal iTRS with an infinite number of rules such that for no rule $l \to r$ does l unify with r; this iTRS contains a weakly convergent reduction that is not strongly convergent.

We now give a number of corollaries showing the usefulness of Theorem 3.8.

Corollary 3.11. *Let R consist of a finite number of rules. If R does not admit a cycle of length* 1*, then every weakly convergent reduction is strongly convergent.*

Corollary 3.12. *Let R consist of a finite number of rules. Then there is a weakly convergent reduction of uncountable length iff R admits a cycle of length* 1*.*

Proof. If R admits a cycle of length 1, there are weakly convergent reductions of any ordinal length. If R does not admit a cycle of length 1, then every weakly convergent reduction is strongly convergent, hence of countable length by Proposition 2.9. ∎

Recall that left-linear systems enjoy the *compression property* in strongly convergent rewriting: If $s \twoheadrightarrow_{\mathrm{S}} t$, then there is a strongly convergent reduction from s to t of length $\leq \omega$. As a curiosity, we mention the following result showing that failure of compression to length at most ω entails existence of a cycle of length 1:

Corollary 3.13. *Let R be a left-linear iTRS with a finite number of rules. If there is a reduction $s \twoheadrightarrow_{W} t$ such that there is* no *reduction from s to t of length at most ω, then $s \twoheadrightarrow_{W} t' \twoheadrightarrow_{W} t$ where $t' \to t'$.*

Proof. $s \twoheadrightarrow_{\mathrm{W}} t$ cannot be strongly convergent (as $s \twoheadrightarrow_{\mathrm{S}} t$ would entail a strongly convergent reduction of length at most ω from s to t by compression). The result then follows from Theorem 3.8. ∎

We can also reason about the possible lengths of weakly convergent reductions depending on whether they admit cycles of length 1:

Corollary 3.14. *Let R be an iTRS having a finite number of rules. Then for any ordinal $\alpha \geq \Omega$, there exists a weakly convergent reduction of length α iff R admits a cycle of length* 1*.*

Proof. If there is a weakly convergent reduction of uncountable length, it cannot be strongly convergent by Proposition 2.9, and Theorem 3.8 proves existence of a cycle of length 1. Conversely, if R admits a cycle of length 1, there are weakly convergent reductions of any ordinal length. ∎

Corollary 3.15. *Let R be an iTRS having a finite number of rules. Then R admits a cycle of length* 1 *iff there is no ordinal α such that all weakly convergent reductions are of length $< \alpha$.*

Proof. If R admits a cycle of length 1, there are weakly convergent reductions of any ordinal length. If R does not admit a cycle of length 1, then by Theorem 3.8, every weakly convergent reduction is strongly convergent, whence Proposition 2.9 yields that every weakly convergent reduction has length $< \Omega$, concluding the proof. ∎

We summarize the results of this section in the following theorem:

Theorem 3.16. *Let R be an iTRS consisting of a finite number of rules. The following are equivalent:*

1. *Every weakly convergent reduction is strongly convergent.*
2. *There does* not *exist a term t with $t \to t$.*
3. *Every weakly convergent reduction has countable length.*
4. *There exists an ordinal α such that all weakly convergent reductions are of length $< \alpha$.*

Proof. $1 \Rightarrow 2$ follows as existence of a term $t \to t$ yields the weakly, but not strongly, convergent reduction $t \to t \to t \to \cdots t$. $2 \Rightarrow 1$ is Corollary 3.11. $2 \Leftrightarrow 4$ is Corollary 3.14, and $2 \Leftrightarrow 3$ is Corollary 3.15. ∎

4. Uniform convergence and uniform normalization

We now show uniform normalization under both weak and strong convergence for orthogonal iTRSs with a finite number of rules; this is the main result of the paper.

Theorem 4.1. *The following are equivalent for an orthogonal iTRS R with a finite number of rules:*

(1) $EXT^{\infty}_{W}(R)$ *and R admits no cycle of length* 1
(2) $SN^{\infty}_{W}(R)$
(3) $WN^{\infty}_{W}(R)$
(4) $WN^{\infty}(R)$
(5) $SN^{\infty}(R)$

Proof. We prove $(2) \Leftrightarrow (1) \Leftrightarrow (5) \Leftrightarrow (4) \Leftrightarrow (3)$.

$(2) \Leftrightarrow (1)$ follows by Theorem 3.16. $(1) \Rightarrow (5)$ follows from Corollary 3.11. For $(5) \Rightarrow (1)$ reason as follows: $SN^{\infty}(R) \Rightarrow EXT^{\infty}_{W}(R)$ follows by noting that $SN^{\infty}(R)$ implies that all maximal reductions end in a normal form [19], and that there are no divergent reductions, hence that all weakly continuous reductions are strongly continuous. If R admited a cycle of length 1, say $s \to s$, then the reduction $s \to s \to s \to \cdots$ of length ω constructed by iterating the step performed in $s \to s$ is strongly continuous, but not strongly convergent, contradicting $SN^{\infty}(R)$. $(4) \Leftrightarrow (5)$ is the content of Theorem 1.1. $(3) \Rightarrow (4)$ follows from Proposition 2.13. $(4) \Rightarrow (3)$ follows by observing that if every term s reduces to a normal form by a strongly convergent reduction, that reduction is also weakly convergent. ∎

Note that Example 3.9 shows that the assumption of finiteness of the rule set cannot be omitted in the implications $((1) \vee (2) \vee (3)) \Rightarrow (4)$, respectively $((1) \vee (2) \vee (3)) \Rightarrow (5)$.

An immediate consequence of Theorem 4.1 is the following confluence result in weakly convergent rewriting:

Corollary 4.2. *Let R be an orthogonal iTRS with a finite number of rules such that at least one of the following holds*

(1) $WN^{\infty}_{W}(R)$
(2) *R admits no cycle of length* 1

Then $CR^{\infty}_{W}(R)$.

Proof. If $WN^{\infty}_{W}(R)$, Theorem 4.1 implies that R admits no cycle of length 1. Thus, it suffices to prove that $CR^{\infty}_{W}(R)$ if R admits no cycle of length 1. By Corollary 3.11, every weakly convergent reduction is also strongly convergent. As R admits no cycle of length 1, R cannot contain a collapsing rule $l \to x$ as it would give rise to a cycle of length 1: Letting the term s be the fixed point of $s = l\{x \mapsto s\}$, we obtain $s \to s$. The result now follows as orthogonal, non-collapsing iTRSs R satisfy $CR^{\infty}(R)$ [11]. ∎

As pointed out by a referee, part (1) of the above corollary can also be proved directly from existing results: If $WN^{\infty}_{\mathrm{W}}(R)$ for an orthogonal iTRS R, then by [19], we have $WN^{\infty}(R)$ and hence $SN^{\infty}(R)$, and as orthogonal iTRSs have unique normal forms under strong convergence, we obtain $CR^{\infty}_{\mathrm{W}}(R)$.

5. Conclusion and conjectures

We have used acyclicity to study the differences between weak and strong convergence in iTRSs R with a finite number of rules. In particular, we established a necessary and sufficient criterion for every weakly convergent reduction to also be strongly convergent: That R admits no cycle of length 1. This criterion was employed to show equivalence of (the infinitary analogues of) weak and strong normalization in both weakly and strongly convergent rewriting. As a further consequence, we have derived new results concerning normalization and confluence in the setting of weakly convergent rewriting.

The results of the paper strongly suggest the following conjecture:

Conjecture 5.1. Let R be an orthogonal iTRSs such that (i) R has a finite set of rules, and (ii) R is almost non-collapsing (that is, R has at most one collapsing rule $l \to x$ and the only variable occurring in l is x). Then, $CR^{\infty}_{\mathrm{W}}(R)$.

If (ii) does not hold, then $CR^{\infty}_{\mathrm{W}}(R)/{\sim_h}$ where $\sim_h$ is equivalence modulo identification of hypercollapsing subterms (see for example [11, 9] for definitions).

The assumption of finiteness of the rule set is crucial in the above conjecture, as witnessed by the counterexamples of [25].

A more modest conjecture that could perhaps be proved more easily is:

Conjecture 5.2. Let R be a non-collapsing iTRS having a finite set of rules. Then $CR^{\infty}_{\mathrm{W}}(R)/{\sim_c}$ where $\sim_c$ is equivalence modulo identification of *cyclic* subterms.

Acknowledgements

The author wishes to thank Patrick Bahr, Jeroen Ketema, Roel de Vrijer, and the anonymous referees for numerous style-improving comments and bug-spotting.

References

[1] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

[2] A. Berarducci and M. V. Zilli. Generalizations of unification. *Journal of Symbolic Computation*, 16(5):479–491, 1993.

[3] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[4] N. Dershowitz, S. Kaplan, and D. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite .... *Theoretical Computer Science*, 83(1):71–96, 1991.

[5] J. Endrullis, C. Grabmayer, D. Hendriks, J. W. Klop, and R. C. de Vrijer. Proving infinitary normalization. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *TYPES*, volume 5497 of *Lecture Notes in Computer Science*, pages 64–82. Springer, 2008.

[6] S. Kahrs. Infinitary rewriting: Meta-theory and convergence. *Acta Informatica*, 44(2):91–121, 2007.

[7] S. Kahrs. Modularity of convergence in infinitary rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA '09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, 2009.

[8] R. Kennaway. On transfinite abstract reduction systems. Technical Report CS-R9205, Centrum voor Wiskunde & Informatica, 1992.
[9] R. Kennaway and F.-J. de Vries. Infinitary rewriting. In *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*, chapter 12, pages 668–711. Cambridge University Press, 2003.
[10] R. Kennaway, J. Klop, M. Sleep, and F.-J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175:93–125, 1997.
[11] R. Kennaway, J. Klop, R. Sleep, and F.-J. de Vries. Transfinite reductions in orthogonal term rewriting. *Information and Computation*, 119(1):18–38, 1995.
[12] R. Kennaway, J. W. Klop, M. R. Sleep, and F.-J. de Vries. Transfinite reductions in orthogonal term rewriting systems (extended abstract). In R. V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1991.
[13] J. Ketema. On normalization of infinitary combinatory reduction systems. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA '08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 2008.
[14] J. Ketema and J. G. Simonsen. Infinitary combinatory reduction systems. In J. Giesl, editor, *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA '05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 438–452. Springer-Verlag, 2005.
[15] J. Ketema and J. G. Simonsen. Infinitary combinatory reduction systems: Confluence. *Logical Methods in Computer Science*, 5(4:3):1–29, 2009.
[16] Z. Khasidashvili, M. Ogawa, and V. van Oostrom. Perpetuality and uniform normalization in orthogonal rewrite systems. *Information and Computation*, 164:118–151, 2001.
[17] Z. Khasidashvili, M. Ogawa, and V. van Oostrom. Uniform normalisation beyond orthogonality. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA '01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 122–136. Springer-Verlag, 2001.
[18] J. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
[19] J. Klop and R. de Vrijer. Infinitary normalization. In *We will show them! Essays in honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.
[20] K. Kunen. *Set Theory: An Introduction to Independence Proofs*, volume 102 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1980.
[21] S. Lucas. Transfinite rewriting semantics for term rewriting systems. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 216–230. Springer-Verlag, 2001.
[22] P. M. Neergaard and M. H. Sørensen. Conservation and uniform normalization in lambda calculi with erasing reductions. *Information and Computation*, 178(1):149 – 179, 2002.
[23] M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1977.
[24] P. Rodenburg. Termination and confluence for infinitary term rewriting. *Journal of Symbolic Logic*, 63(4):1286–1296, 1998.
[25] J. G. Simonsen. On confluence and residuals in Cauchy convergent transfinite rewriting. *Information Processing Letters*, 91(3):141–146, 2004.
[26] H. Xi. Weak and strong beta normalisations in typed λ-calculi. In *In: Proc. of the 3rd International Conference on Typed Lambda Calculus and Applications, TLCA'97*, pages 390–404. Springer Verlag, 1997.
[27] H. Zantema. Normalization of infinite terms. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA '08)*, volume 5117 of *Lecture Notes in Computer Science*, pages 441–455. Springer-Verlag, 2008.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 325-340
http://rewriting.loria.fr/rta/

CERTIFIED SUBTERM CRITERION AND CERTIFIED USABLE RULES

CHRISTIAN STERNAGEL [1] AND RENÉ THIEMANN [1]

[1] Institute of Computer Science, University of Innsbruck, Austria

E-mail address: christian.sternagel@uibk.ac.at
E-mail address: rene.thiemann@uibk.ac.at
URL: http://cl-informatik.uibk.ac.at/software/ceta

Abstract. In this paper we present our formalization of two important termination techniques for term rewrite systems: the subterm criterion and the reduction pair processor in combination with usable rules. For both techniques we developed executable check functions using the theorem prover Isabelle/HOL. These functions are able to certify the correct application of the formalized techniques in a given termination proof. As there are several variants of usable rules, we designed our check function in such a way that it accepts all known variants, even those which are not explicitly spelled out in previous papers.

We integrated our formalization in the publicly available IsaFoR-library. This led to a significant increase in the power of CeTA, a certified termination proof checker that is extracted from IsaFoR.

1. Introduction

Termination provers for term rewrite systems (TRSs) became more and more powerful in the last years. One reason is that a proof of termination no longer is just some reduction order which contains the rewrite relation of the TRS. Currently, most provers construct a proof in the dependency pair framework (DP framework). This allows to combine basic termination techniques in a flexible way. Hence, a termination proof is a tree where at each node a specific technique is applied. So instead of just stating the precedence of some lexicographic path order or giving some polynomial interpretation, current termination provers return proof trees consisting of many different techniques and reaching sizes of several megabytes. Thus, it would be too much work to check by hand whether these trees really form a valid proof. (Additionally, checking by hand does not provide a very high degree of confidence.)

It is regularly demonstrated that we cannot blindly trust in the output of termination provers. Every now and then, some termination prover delivers a faulty proof. Most often, this is only detected if there is another prover giving a contradicting answer on the same problem. To solve this problem, three systems have been developed over the last few years: CiME/Coccinelle [4, 5], Rainbow/CoLoR [3], and CeTA/IsaFoR [23]. These systems either certify or reject a given termination proof. Here, Coccinelle and CoLoR

This research is supported by FWF (Austrian Science Fund) project P18763.

© C. Sternagel and R. Thiemann
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.325

are libraries on rewriting for Coq (`http://coq.inria.fr`) and IsaFoR is our library on rewriting for Isabelle [21]. (Throughout this paper we just write Isabelle whenever we refer to Isabelle/HOL.)

All of these certifiers can automatically certify termination proofs that are performed within the DP framework. In this framework one tries to simplify so called DP problems $(\mathcal{P}, \mathcal{R})$ by processors until all pairs in $\mathcal{P}$ are removed.

The reduction pair processor [12, 14] is the major technique to remove pairs. Consequently, it has been formalized in all three libraries. One of the conditions of the processor demands that all rules in $\mathcal{R}$ must be weakly decreasing. If this and all other conditions are satisfied then one can remove all strictly decreasing pairs. In this paper, we present the details about the formalization of two important extensions of the reduction pair processor.

The first extension is the subterm criterion [15]. By restricting the used "reduction pair" to the subterm relation in combination with simple projections, it is possible to ignore the $\mathcal{R}$-component of a DP problem. Note that the subterm criterion has independently (and only recently) been formalized for the Coccinelle-library [4]. Here, we present the first Isabelle formalization of this important technique.

The other extension is the integration of usable rules [9, 10, 12, 14, 24]. With this extension not all rules in $\mathcal{R}$ have to be weakly decreasing but only the usable rules which are most often a strict subset of $\mathcal{R}$. However, there are several definitions of usable rules where the most powerful ones ([10] and [12]) are incomparable.

$$\text{all rules} \;\supseteq\; \text{usable rules ([24])} \;\supseteq\; \text{usable rules ([9, 14])} \begin{array}{l} \rotatebox{45}{$\supseteq$} \; \text{usable rules ([10])} \\ \rotatebox{-45}{$\supseteq$} \; \text{usable rules ([12])} \end{array}$$

Although it was often stated that a combination of the definitions of usable rules of [10] and [12] would be possible there never was a refereed paper which showed such a proof. (However, there have been unpublished soundness proofs of such a combined definition.) In this paper we not only present such a combined definition and the first corresponding *formalized* soundness proof, but we also simplified and extended the existing proofs. For example, we never construct filtered terms although we consider usable rules w.r.t. some argument filter. (An independent formalization of usable rules is present in Coccinelle. However, this formalization is unpublished and it only uses the variant of [14]: it does not feature the improvements from [10] and [12].) With these two extensions of the reduction pair processor we could increase the number of TRSs (from the Termination Problem Database) where a proof can be certified by our certifier CeTA by over 50%.

Note that all the proofs that are presented (or omitted) in the following, have been formalized in our Isabelle library IsaFoR. Hence, in the paper we merely give sketches of our "real" proofs. Our goal is to show the general proof outlines and help to understand the full proofs. Our library IsaFoR with all formalized proofs, the executable certifier CeTA, and all details about our experiments are available at CeTA's website:

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

The paper is structured as follows: In Sec. 2, we recapitulate the required notions and notations of term rewriting and the DP framework. In Sec. 3, we describe our formalization of the subterm criterion. The reduction pair processor with usable rules and its formalization is presented in Sec. 4. Then, in Sec. 5, we shortly describe how CeTA is obtained from IsaFoR and give a summary about our experiments. We finally conclude in Sec. 6.

2. Preliminaries

Term Rewriting. We assume familiarity with term rewriting [2]. Still, we recall the most important notions that are used later on. A *(first-order) term* t over a set of *variables* $\mathcal{V}$ and a set of *function symbols* $\mathcal{F}$ is either a variable $x \in \mathcal{V}$ or an n-ary function symbol $f \in \mathcal{F}$ applied to n argument terms $f(\vec{t_n})$. A *context* C is a term containing exactly one occurrence of the special constant $\Box$ (that is assumed to be distinct from symbols in $\mathcal{F}$). Replacing $\Box$ in a context C by a term t is denoted by $C[t]$. A term t is a *(proper) subterm* of a term s—written ($s \rhd t$) $s \unrhd t$—whenever there exists a context C ($\neq \Box$), such that $s = C[t]$. We write $s \approx t$ iff s and t are unifiable. An *argument filter* π is a mapping from symbols to integers or lists of integers. It induces a mapping from terms to terms where $\pi(x) = x$, $\pi(f(\vec{t_n})) = \pi(t_i)$ if $\pi(f) = i$, and $\pi(f(\vec{t_n})) = f(\pi(t_{i_1}), \ldots, \pi(t_{i_k}))$ if $\pi(f) = [i_1, \ldots, i_k]$. Argument filters are also used to indicate which positions in a term are regarded. Then π maps symbols to sets of positions. It will be clear from the context which kind of argument filters is used.

A *rewrite rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of rewrite rules. The *rewrite relation (induced by $\mathcal{R}$)* $\to_{\mathcal{R}}$ is the closure under substitutions and under contexts of $\mathcal{R}$, i.e., $s \to_{\mathcal{R}} t$ iff there is a context C, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. Reductions at the root are denoted by $\to_{\mathcal{R},\epsilon}$.

We say that an element t is *terminating/strongly normalizing (w.r.t. some binary relation S)*, and write $\mathsf{SN}_S(t)$, if it cannot start an infinite sequence

$$t = t_1 \; S \; t_2 \; S \; t_3 \; S \cdots .$$

The whole relation is terminating, written $\mathsf{SN}(S)$, if all elements are terminating w.r.t. it. For a TRS $\mathcal{R}$ and a term t, we write $\mathsf{SN}(\mathcal{R})$ and $\mathsf{SN}_{\mathcal{R}}(t)$ instead of $\mathsf{SN}(\to_{\mathcal{R}})$ and $\mathsf{SN}_{\to_{\mathcal{R}}}(t)$. We write S^+ for the transitive closure of S, and S^* is the reflexive transitive closure.

Lemma 2.1 (Properties of Subterms)**.**

(a) *stability:* $s \rhd t \Longrightarrow s\sigma \rhd t\sigma$
(b) *subterms preserve termination:* $\mathsf{SN}_{\mathcal{R}}(s) \wedge s \rhd t \Longrightarrow \mathsf{SN}_{\mathcal{R}}(t)$. ■

Let $\to_{\mathsf{SN}(\mathcal{R})}$ denote the restriction of $\to_{\mathcal{R}}$ to terminating terms, i.e., $\{(s,t) \mid s \to_{\mathcal{R}} t \wedge \mathsf{SN}_{\mathcal{R}}(s)\}$. Let $\overset{\rhd}{\to}_{\mathsf{SN}(\mathcal{R})}$ denote the same relation extended by the restriction of $\rhd$ to terminating terms, i.e., $\to_{\mathsf{SN}(\mathcal{R})} \cup \{(s,t) \mid s \rhd t \wedge \mathsf{SN}_{\mathcal{R}}(s)\}$.

Lemma 2.2 (Termination Properties)**.** *Let S be some binary relation, let $\mathcal{R}$ be a TRS.*

(a) $\mathsf{SN}(S) \Longleftrightarrow \mathsf{SN}(S^+)$,
(b) $\mathsf{SN}(\overset{\rhd}{\to}_{\mathsf{SN}(\mathcal{R})})$,
(c) $\mathsf{SN}_S(s) \wedge (s,t) \in S \Longrightarrow \mathsf{SN}_S(t)$. ■

Dependency Pair Framework. The DP framework [12] is a way to modularize termination proofs. Therefore, we switch from TRSs to so called DP problems, consisting of two TRSs. The *initial DP problem* for a TRS $\mathcal{R}$ is $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$ where $\mathsf{DP}(\mathcal{R})$ are the *dependency pairs* of $\mathcal{R}$. A $(\mathcal{P},\mathcal{R})$*-chain* is a possibly infinite derivation of the following form:

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to_{\mathcal{R}}^* s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to_{\mathcal{R}}^* s_3\sigma_3 \to_{\mathcal{P}} \cdots \qquad (\star)$$

where $s_i \to t_i \in \mathcal{P}$ for all $i > 0$ (this implies that $\mathcal{P}$-steps only occur at the root). If additionally every $t_i\sigma_i$ is terminating w.r.t. $\mathcal{R}$, then the chain is *minimal*. A DP problem

$(\mathcal{P},\mathcal{R})$ is called *finite* [12], if there is no minimal $(\mathcal{P},\mathcal{R})$-chain. Proving finiteness of a DP problem is done by simplifying $(\mathcal{P},\mathcal{R})$ by so called *processors* recursively, until the $\mathcal{P}$-components of all remaining DP problems are empty and therefore trivially finite. For this to be correct, the applied processors need to be *sound*. A processor *Proc* is sound whenever for all DP problems $(\mathcal{P},\mathcal{R})$ we have that finiteness of $(\mathcal{P}',\mathcal{R}')$ for all $(\mathcal{P}',\mathcal{R}') \in \mathit{Proc}(\mathcal{P},\mathcal{R})$ implies finiteness of $(\mathcal{P},\mathcal{R})$. The termination techniques that will be introduced in the following sections are all such sound processors.[1]

Example 2.3. In the following TRS $\mathcal{R}$ the term $\mathsf{set}(xs)$ evaluates to the list $[x \in xs \mid 0 < x]$ where duplicates are removed:

$$x < \mathsf{0} \to \bot, \quad (2.1) \qquad \mathsf{del}(x,\mathsf{nil}) \to \mathsf{nil}, \quad (2.4)$$

$$\mathsf{0} < \mathsf{s}(y) \to \top, \quad (2.2) \qquad \mathsf{del}(x, y : z) \to \mathsf{if}(x < y, y < x, x, y, z), \quad (2.5)$$

$$\mathsf{s}(x) < \mathsf{s}(y) \to x < y, \quad (2.3) \qquad \mathsf{if}(\bot,\bot,x,y,z) \to \mathsf{del}(x,z), \quad (2.6)$$

$$\mathsf{set}(\mathsf{nil}) \to \mathsf{nil}, \qquad \mathsf{if}(\top,b,x,y,z) \to y : \mathsf{del}(x,z), \quad (2.7)$$

$$\mathsf{set}(x : z) \to \mathsf{if2}(\mathsf{0} < x, x, z), \qquad \mathsf{if}(b,\top,x,y,z) \to y : \mathsf{del}(x,z), \quad (2.8)$$

$$\mathsf{if2}(\top,x,z) \to x : \mathsf{set}(\mathsf{del}(x,z)),$$

$$\mathsf{if2}(\bot,x,z) \to \mathsf{set}(z).$$

After computing the initial DP problem $(\mathsf{DP}(\mathcal{R}),\mathcal{R})$ we can split it into the three problems $(\{(2.9)\},\mathcal{R})$, $(\{(2.13)–(2.16)\},\mathcal{R})$, and $(\{(2.10)–(2.12)\},\mathcal{R})$. (This is done by applying the dependency graph processor [1, 10, 12, 14], a well-known technique to perform separate termination proofs for each recursive function.)

$$\mathsf{s}(x) <^\sharp \mathsf{s}(y) \to x <^\sharp y, \quad (2.9) \qquad \mathsf{del}^\sharp(x, y : z) \to \mathsf{if}^\sharp(x < y, y < x, x, y, z), \quad (2.13)$$

$$\mathsf{if}^\sharp(\bot,\bot,x,y,z) \to \mathsf{del}^\sharp(x,z), \quad (2.14)$$

$$\mathsf{set}^\sharp(x : z) \to \mathsf{if2}^\sharp(\mathsf{0} < x, x, z), \quad (2.10) \qquad \mathsf{if}^\sharp(\top,b,x,y,z) \to \mathsf{del}^\sharp(x,z), \quad (2.15)$$

$$\mathsf{if2}^\sharp(\top,x,z) \to \mathsf{set}^\sharp(\mathsf{del}(x,z)), \quad (2.11) \qquad \mathsf{if}^\sharp(b,\top,x,y,z) \to \mathsf{del}^\sharp(x,z). \quad (2.16)$$

$$\mathsf{if2}^\sharp(\bot,x,z) \to \mathsf{set}^\sharp(z), \quad (2.12)$$

3. The Subterm Criterion

The subterm criterion [15] is a termination technique that can be employed as a processor of the DP framework. It may be seen as a variant of the reduction pair processor with an attached argument filtering [1]. The used orders ($\rhd$ and $\unrhd$) allow to ignore the $\mathcal{R}$ component of a DP problem $(\mathcal{P},\mathcal{R})$. And the argument filtering is restricted to be a so called *simple projection*. A simple projection π maps a term to one of its arguments, i.e., $\pi(f(\vec{t_n})) = t_i$ for some $0 < i \leqslant n$. For convenience we use R_π to denote the 'composition' of the binary relation on terms R and π, i.e., $(s,t) \in R_\pi$ iff $(\pi(s),\pi(t)) \in R$.

Theorem 3.1. *Finiteness of $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$ implies finiteness of $(\mathcal{P},\mathcal{R})$, provided:*

[1]To be more precise, in IsaFoR it is shown that all these processors are chain identifying (`chain_identifying_proc`) which is a slightly stronger requirement than soundness [22, Chapter 7]. The reason is that chain identifying processors can easily be combined with semantic labeling [25]. However, we omit the details here and just refer to theory `DpFramework` for the interested reader.

(a) *all rules of* $\mathcal{P}$ *are oriented by* $\trianglerighteq_\pi$ *(i.e.,* $\mathcal{P} \subseteq \trianglerighteq_\pi$*)*
(b) *all lhss and rhss of* $\mathcal{P}$ *are non-variable and non-constant terms where the roots of rhss are not defined in* $\mathcal{R}$ *(i.e.,* $s = f(\vec{s}_n)$ *with* $n > 0$ *and* $t = g(\vec{t}_m)$ *with* $m > 0$ *and* $g \notin \mathcal{D}_\mathcal{R}$ *for all* $s \to t \in \mathcal{P}$*)*

Example 3.2. The DP problem $(\{(2.9)\}, \mathcal{R})$ from Ex. 2.3 can be solved using the simple projection $\pi(<^\sharp) = 1$, since $\pi(\mathsf{s}(x) <^\sharp \mathsf{s}(y)) = \mathsf{s}(x) \rhd x = \pi(x <^\sharp y)$. Taking $\pi(\mathsf{del}^\sharp) = 2$ and $\pi(\mathsf{if}^\sharp) = 5$ we can remove Pair (2.13) from $(\{(2.13)\text{–}(2.16)\}, \mathcal{R})$. The result $(\{(2.14)\text{–}(2.16)\}, \mathcal{R})$ is then solved by the dependency graph processor. Removing a pair from $(\{(2.10)\text{–}(2.12)\}, \mathcal{R})$ is impossible as there is no π such that Pair (2.11) is oriented. Note that $<^\sharp, \mathsf{del}^\sharp, \mathsf{if}^\sharp, \ldots \notin \mathcal{D}_\mathcal{R}$ whereas $<, \mathsf{del}, \mathsf{if}, \dots \in \mathcal{D}_\mathcal{R}$.

Before we can prove Theorem 3.1, we need several lemmas. First, we prove that termination of some element w.r.t. some binary relation S is equivalent to termination of the same element w.r.t. S^+. Note that this is a more general result than Lem. 2.2(a) and thus allows termination analysis of a single term, no matter if the whole TRS is terminating.

Lemma 3.3. $\mathsf{SN}_S(t) \iff \mathsf{SN}_{S^+}(t)$.

Proof. The direction from right to left is trivial. For the other direction assume that t is not terminating w.r.t. S^+. Hence $t = t_1\ S^+\ t_2\ S^+\ t_3\ S^+ \cdots$. Let S' denote the restriction of R to terminating terms, i.e., $S' = \{(s,t) \mid s\ S\ t \land \mathsf{SN}_S(s)\}$. By definition we have $\mathsf{SN}(S')$ and with Lem. 2.2(a) also $\mathsf{SN}(S'^+)$. Using $\mathsf{SN}_S(t)$ and Lem. 2.2(c) together with the infinite sequence from above, we get $\mathsf{SN}_S(t_i)$ for all $i > 0$, and further $t_1\ S'^+\ t_2\ S'^+\ t_3\ S'^+ \cdots$. This contradicts $\mathsf{SN}(S'^+)$. ∎

Next consider a general result on infinite sequences conducted in the union of two binary relations N and S where often N is a non-strict relation and S a strongly normalizing relation. Intuitively it states the following: Assume that there is an infinite sequence of steps, where each step is an N-step or an S-step. Further assume that whenever there is an N-step directly followed by an S-step, those two steps can be turned into a single S-step. Additionally, there is no infinite S-sequence starting at the same point as the sequence we are reasoning about. Then, from some point in our sequence on, there are no more S-steps, i.e., it ends in N-steps. This is a versatile fact that is used at several places inside IsaFoR.

Lemma 3.4. *Let* N *and* S *be two binary relations over some carrier and* $\vec{q}$ *an infinite sequence of carrier elements. If*
(a) $(q_i, q_{i+1}) \in N \cup S$ *for all* $i > 0$*,*
(b) $N \circ S \subseteq S$*, and*
(c) $\mathsf{SN}_S(q_1)$*,*

then there is some j *such that for all* $i \geq j$ *we have* $(q_i, q_{i+1}) \in N \setminus S$*.*

Proof. For the sake of a contradiction assume that the lemma does not hold. Then, together with (a), we obtain $\forall i > 0.\, \exists j \geq i.\, (q_j, q_{j+1}) \in S$. Using the *Axiom of Choice* we get hold of a choice function f such that

$$\forall i > 0.\, f(i) \geq i \land (q_{f(i)}, q_{f(i)+1}) \in S, \qquad (\dagger)$$

i.e., $f(i)$ produces some index of an S-step after position i in $\vec{q}$. Using f we define a new sequence $[\cdot]$ of indices inductively

$$[i] = \begin{cases} i & \text{if } i = 1, \\ f([i-1]) + 1 & \text{otherwise.} \end{cases}$$

With (†) we have $f(i) \geq i$ and $(q_{f(i)}, q_{f(i)+1}) \in S$ for all $i > 0$. Since $f(i) \geq i$ there is an $N \cup S$ sequence from every q_i to the corresponding $q_{f(i)}$. Thus we obtain $(q_i, q_{f(i)+1}) \in S^+$ for all $i > 0$ using (b). This immediately implies $(q_{[i]}, q_{[i+1]}) \in S^+$ for all $i > 0$ and thereby $\neg\mathsf{SN}_{S^+}(q_{[1]})$ which is equivalent to $\neg\mathsf{SN}_S(q_{[1]})$ by Lem. 3.3. But $q_{[1]} = q_1$ and thus $\neg\mathsf{SN}_S(q_1)$. Together with (c), this provides the desired contradiction. ∎

Lemma 3.5. $\mathsf{SN}_{\mathcal{R}}(t) \Longrightarrow \mathsf{SN}_{(\rhd\cup\rightarrow_{\mathcal{R}})}(t)$.

Proof. Assume that t is not terminating w.r.t. $(\rhd \cup \rightarrow_{\mathcal{R}})$. Hence, we obtain the infinite sequence $t = t_1\ (\rhd \cup \rightarrow_{\mathcal{R}})\ t_2\ (\rhd \cup \rightarrow_{\mathcal{R}})\ t_3\ (\rhd \cup \rightarrow_{\mathcal{R}}) \cdots$. From the assumption we have $\mathsf{SN}_{\mathcal{R}}(t_1)$ and by Lem. 2.1 and Lem. 2.2(c) we obtain $\mathsf{SN}_{\mathcal{R}}(t_i)$ for all $i > 0$. Thus, $t_i \stackrel{\rhd}{\rightarrow}_{\mathsf{SN}(\mathcal{R})} t_{i+1}$ for all i and since $\mathsf{SN}(\stackrel{\rhd}{\rightarrow}_{\mathsf{SN}(\mathcal{R})})$ by Lem. 2.2(b) we arrive at a contradiction. ∎

Proof of Theorem 3.1. In order to show that finiteness of $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$ we prove its contraposition. Hence, we may assume (in addition to the premises of Theorem 3.1) that there is a minimal infinite $(\mathcal{P}, \mathcal{R})$-chain and have to transform it into a minimal infinite $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$-chain. Thus we may assume that for all $i > 0$:

(a) $s_i \rightarrow t_i \in \mathcal{P}$,
(b) $t_i\sigma_i \rightarrow^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$, and
(c) $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$.

We start by a case distinction on $\exists j > 0. \forall i \geq j. (s_i, t_i) \in (\mathcal{P} \setminus \rhd_\pi)$. If there is such a j, we can combine this with (b) and obtain the desired minimal $(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})$-chain by shifting the original chain j positions to the left. Hence, consider the second case and assume $\forall i > 0. \exists j \geq i. (s_j, t_j) \notin (\mathcal{P} \setminus \rhd_\pi)$. With (a) and the preconditions of the subterm criterion processor this results in

$$\forall i > 0. \exists j \geq i. \pi(s_j)\sigma_j \rhd \pi(t_j)\sigma_j. \tag{3.1}$$

From this point on, the proof mainly runs by instantiating the relations N and S of Lem. 3.4 appropriately and showing the assumptions Lem. 3.4(a)–Lem. 3.4(c) in turn. For N we use the reflexive and transitive closure of the rewrite relation, i.e., $\rightarrow^*_{\mathcal{R}}$. For S we use $(\rhd \cup \rightarrow_{\mathcal{R}})^+$. Finally, we use the infinite sequence q defined by $q_i = \pi(s_{i+1})\sigma_{i+1}$ (the index shift is needed to establish termination of q_1 later on). From (a) and Thm. 3.1(a), together with Lem. 2.1(a) we get

$$\pi(s_i)\sigma_i \trianglerighteq \pi(t_i)\sigma_i. \tag{3.2}$$

Furthermore, we obtain

$$\pi(t_i)\sigma_i \rightarrow^*_{\mathcal{R}} \pi(s_{i+1})\sigma_{i+1}, \tag{3.3}$$

since the roots of t_i and s_{i+1} are guaranteed to be non-constant symbols and the root of t_i is not a defined symbol by Thm. 3.1(b). In combination we get $\pi(s_i)\sigma_i \trianglerighteq \circ \rightarrow^*_{\mathcal{R}} \pi(s_{i+1})\sigma_{i+1}$ and in turn $(q_i, q_{i+1}) \in N \cup S$, thereby discharging assumption Lem. 3.4(a). For our specific relations assumption Lem. 3.4(b) trivially holds. This leaves us with showing termination of q_1 with respect to the relation $(\rhd \cup \rightarrow_{\mathcal{R}})^+$. From the minimality of the initial chain (c) we know $\mathsf{SN}_{\mathcal{R}}(t_1\sigma_1)$ and by Lem. 2.1 we get $\mathsf{SN}_{\mathcal{R}}(\pi(s_2)\sigma_2)$ and thus $\mathsf{SN}_{\mathcal{R}}(q_1)$. By Lemmas 3.5 and 3.3 we then achieve $\mathsf{SN}_{(\rhd\cup\rightarrow_{\mathcal{R}})^+}(q_1)$. At this point (by Lem. 3.4) we get grip of some $j > 0$ such that

$$\forall i \geq j. (q_i, q_{i+1}) \in N \setminus S. \tag{3.4}$$

Now we proof $\forall i \geq j. \pi(s_{i+1})\sigma_{i+1} = \pi(t_{i+1})\sigma_{i+1}$ as follows. Assume $i \geq j$ and $\pi(s_{i+1})\sigma_{i+1} \neq \pi(t_{i+1})\sigma_{i+1}$. Then with (3.2) we get $\pi(s_{i+1})\sigma_{i+1} \rhd \pi(t_{i+1})\sigma_{i+1}$. By (3.3), this results in

$\pi(s_{i+1})\sigma_{i+1} \rhd \circ \to_{\mathcal{R}}^{*} \pi(s_{i+2})\sigma_{i+2}$ and consequently in $(q_i, q_{i+1}) \in S$ (contradicting 3.4). Thus $\forall i \geq j.\, q_i = \pi(t_{i+1})\sigma_{i+1}$. However, this contradicts (3.1). ∎

4. Usable Rules

One important technique to prove termination within the DP framework is the reduction pair processor. A *reduction pair* $(\succ, \succsim)$ consists of a well-founded and stable relation $\succ$ in combination with a monotone and stable relation $\succsim$. Further, $\succsim$ has to be compatible with $\succ$, i.e., $\succsim \circ \succ \subseteq \succ$. Note that it is not required that $\succ$ and $\succsim$ are partial orders [23]. Examples for reduction pairs are polynomial orders [15, 19, 20], matrix orders [7, 17], and the lexicographic path order (LPO) [16]. (There are several other classes of reduction pairs. We listed those which have been formalized in IsaFoR.)

The basic version of the reduction pair processor [12, 14] requires that all rules of $\mathcal{R}$ are weakly decreasing w.r.t. $\succsim$ (then $\to_{\mathcal{R}} \subseteq \succsim$) and all pairs of $\mathcal{P}$ are weakly or strictly decreasing. From ($\star$) on page 327 it is easy to see that this implies that every reduction in a $(\mathcal{P}, \mathcal{R})$-chain corresponds to a weak or strict decrease. Thus, the strictly decreasing pairs cannot occur infinitely often and can be removed from $\mathcal{P}$. This technique is already present in IsaFoR and its formalization is described in [23].

Theorem 4.1. *Finiteness of* $(\mathcal{P} \setminus \succ, \mathcal{R})$ *implies finiteness of* $(\mathcal{P}, \mathcal{R})$*, provided:*
(a) $(\succ, \succsim)$ *is a reduction pair,*
(b) $\mathcal{P} \subseteq \succ \cup \succsim$*,*
(c) $\mathcal{R} \subseteq \succsim$*.*

Starting with [24], there have been several papers [10, 12, 14] on how to improve the last requirement. Therefore, $\mathcal{R}$ in (c) is replaced by the *usable rules.*

The main idea of the usable rules is easy to explain: since in chains rewriting is only performed with instances of rhss of $\mathcal{P}$, it should suffice to rewrite with rules of defined symbols that occur in rhss of $\mathcal{P}$. If these usable rules introduce new defined symbols then the rules defining them also have to be considered as usable. Hence, in the remaining DP problem $(\{(2.10)\text{–}(2.12)\}, \mathcal{R})$ of Ex. 2.3 only rules (2.1)–(2.3) and (2.4)–(2.8) are usable. This idea is formally expressed in the following definition.

Definition 4.2 (Usable Rules)**.** The function $\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t)$ defines whether a term t is closed under usable rules $\mathcal{U}$ w.r.t. some TRS $\mathcal{R}$.

$$\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(x) = \mathsf{true},$$

$$\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(f(\vec{t_n})) = \bigwedge_{1 \leqslant i \leqslant n} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t_i) \wedge \bigwedge_{\ell \to r \in \mathcal{R}} (\mathsf{root}(\ell) = f \Longrightarrow \ell \to r \in \mathcal{U}).$$

A TRS $\mathcal{Q}$ is closed under usable rules whenever all rhss are closed under usable rules, i.e.,

$$\mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(\mathcal{Q}) = \bigwedge_{\ell \to r \in \mathcal{Q}} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(r).$$

A DP problem $(\mathcal{P}, \mathcal{R})$ is closed under usable rules iff $\mathcal{P}$ and $\mathcal{U}$ are closed under usable rules.

$$\mathsf{urClosed}_{\mathcal{U}}(\mathcal{P}, \mathcal{R}) = \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(\mathcal{P}) \wedge \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(\mathcal{U}).$$

Finally, the usable rules of DP problem $(\mathcal{P}, \mathcal{R})$ are the least set $\mathcal{U}$ satisfying $\mathsf{urClosed}_{\mathcal{U}}(\mathcal{P}, \mathcal{R})$.

Note that there are several other equivalent definitions of usable rules, e.g., one can find definitions of usable rules via so called usable symbols (i.e., root symbols of lhss of usable rules). However, there are also two improvements that yield smaller sets.[2]

The first improvement is to take the reduction pair into account. If certain positions of terms are disregarded then their usable rules do not have to be considered. For example, usually due to the rhs $\mathsf{if2}^\sharp(\mathsf{0} < x, x, z)$ of DP (2.10) all $<$-rules are usable. However, if we would use a reduction pair based on polynomial orders, where $\mathcal{P}ol(\mathsf{if2}^\sharp(b, x, z)) = z$ then the call to $<$ is ignored by the order. This can be exploited by excluding the $<$-rules from the set of usable rules. This improvement is called *usable rules w.r.t. an argument filter* [12]. Here, argument filters are used to describe which positions in a term are relevant: an argument filter π with $\pi(f) = \{i_1, \dots, i_k\}$ indicates that in a term $f(\vec{t_n})$ only arguments $t_{i_1}, \dots, t_{i_k}$ are regarded. This is formalized by the notion of π-compatibility.

Definition 4.3 (π-Compatibility)**.** A relation $\succsim$ is π*-compatible* iff for all n-ary symbols f, all i with $1 \leqslant i \leqslant n$, all $t_1, \dots, t_n$, and all s and s':

$$i \notin \pi(f) \Longrightarrow f(t_1, \dots, t_{i-1}, s, t_{i+1}, \dots, t_n) \succsim f(t_1, \dots, t_{i-1}, s', t_{i+1}, \dots, t_n).$$

To formally define the usable rules w.r.t. an argument filter, a minor modification of Def. 4.2 suffices. Just

$$\text{replace} \quad \bigwedge_{1 \leqslant i \leqslant n} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t_i) \quad \text{by} \quad \bigwedge_{i \in \pi(f)} \mathsf{urClosed}_{\mathcal{U},\mathcal{R}}(t_i).$$

The second improvement to reduce the set of usable rules is performed by taking the structure of terms into account. Observe that in the rhs $\mathsf{if2}^\sharp(\mathsf{0} < x, x, z)$ of DP (2.10), the first argument of $<$ is $\mathsf{0}$. Hence, only the rules (2.1) and (2.2) are possibly applicable, but not the remaining Rule (2.3). This is not captured by Def. 4.2. As there, all f-rules have to be usable whenever the symbol f occurs. On the contrary, in [10], an improved version of usable rules is described which can figure out that Rule (2.3) is not applicable. To this end, the condition $\mathsf{root}(\ell) = f$ in Def. 4.2 is replaced by a condition based on unification. Demanding $\ell \approx f(\vec{t_n})$ would be unsound. First $f(\vec{t_n})$ has to be preprocessed by the function tcap of [10]. This function keeps only those parts of the input term which cannot be reduced, even if the term is instantiated. All other parts are replaced by fresh variables.

Definition 4.4. Let $\mathcal{R}$ be a TRS.[3]

$$\mathsf{tcap}(t) = \begin{cases} f(\mathsf{tcap}(\vec{t_n})) & \text{if } t = f(\vec{t_n}) \text{ and } \ell \not\approx f(\mathsf{tcap}(\vec{t_n})) \text{ for all lhss } \ell \text{ of } \mathcal{R}, \\ \text{a fresh variable} & \text{otherwise.} \end{cases}$$

Here, $\mathsf{tcap}(\vec{t_n})$ is the list of terms where tcap is applied to all arguments of $\vec{t_n}$.

[2]There also is another extension of usable rules, the *generalized usable rules*. It allows a variant of the reduction pair processor where reduction pairs with non-monotone $\succsim$ may be used, cf. [8, Thm. 10]. However, that reduction pair processor is incomparable to both Thm. 4.1 and the upcoming Thm. 4.6. It may be interesting to formalize the soundness of that processor, too, but that would be a different proof.

[3]Note that in IsaFoR we do not use tcap, but the more efficient and equivalent version etcap which is based on ground-contexts. Moreover, unification is replaced by ground-context matching [23, Section 4.2]. But as the notions of tcap and unification are more common, in the following we stick to these two notions.

Now the second improvement can be defined formally. Again, it is a minor but crucial modification of Def. 4.2. It suffices to

$$\text{replace} \bigwedge_{\ell \to r \in \mathcal{R}} (\mathsf{root}(\ell) = f \implies \ell \to r \in \mathcal{U}) \quad \text{by} \bigwedge_{\ell \to r \in \mathcal{R}} (\ell \approx f(\mathsf{tcap}(\vec{t_n})) \implies \ell \to r \in \mathcal{U}).$$

Hence, incorporating both improvements results in the following definition which now contains a π in the superscript to distinguish it from Def. 4.2.

Definition 4.5 (Improved Closure Under Usable Rules)**.**

$$\begin{aligned}
\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(x) &= \mathsf{true}, \\
\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(f(\vec{t_n})) &= \bigwedge_{i \in \pi(f)} \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(t_i) \wedge \bigwedge_{\ell \to r \in \mathcal{R}} (\ell \approx f(\mathsf{tcap}(\vec{t_n})) \Longrightarrow \ell \to r \in \mathcal{U}), \\
\mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{Q}) &= \bigwedge_{\ell \to r \in \mathcal{Q}} \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(r), \\
\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R}) &= \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{P}) \wedge \mathsf{urClosed}^{\pi}_{\mathcal{U},\mathcal{R}}(\mathcal{U}).
\end{aligned}$$

Note that we do not define *the* improved usable rules of a DP problem $(\mathcal{P},\mathcal{R})$ (as we would, by demanding that $\mathcal{U}$ is the least set satisfying $\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$). Hence, every set $\mathcal{U}$ that satisfies the closure properties can be used later on. It is easy to see, that the usable rules w.r.t. Def. 4.2 satisfy the closure properties as well as a version of usable rules which only incorporates one of the two improvements. Thus, by this definition we gain the advantage that we can handle several variants of usable rules.

Having defined all necessary notions, we are ready to present the improved reduction pair processor with usable rules where the second part also incorporates [9, Theorem 28] which allows to delete rules by a syntactic criterion.

Theorem 4.6 (Reduction Pair Processors with Usable Rules)**.** *Let* c *be some binary symbol, let* $\mathcal{C}_\varepsilon = \{\mathsf{c}(x,y) \to x, \mathsf{c}(x,y) \to y\}$,[4] *and let* $\mathcal{U}$ *be some TRS (called the usable rules). For every signature* $\mathcal{F}$ *and TRS* $\mathcal{R}$*, let* $\mathcal{R}_{\neg\mathcal{F}} = \{\ell \to r \in \mathcal{R} \mid \ell \notin \mathcal{T}(\mathcal{F},\mathcal{V})\}$*. Finiteness of* $(\mathcal{P} \setminus \succ, \mathcal{R})$ *implies finiteness of* $(\mathcal{P},\mathcal{R})$*, provided:*

(a) $(\succ, \succsim)$ *is a reduction pair,*
(b) $\mathcal{P} \subseteq \succ \cup \succsim$,
(c) $\mathcal{U} \cup \mathcal{C}_\varepsilon \subseteq \succsim$,
(d) $\succsim$ *is* π*-compatible,*
(e) $\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$.

If additionally $\mathcal{C}_\varepsilon \subseteq \succ$, $\succ$ *is monotone,* $\mathcal{U} \subseteq \mathcal{R}$*, and* $\mathcal{F}$ *is the set of all symbols occurring in rhss of* $\mathcal{P} \cup \mathcal{U}$*, then* $\mathcal{R}$ *can be replaced by* $\mathcal{U}$ *without any strictly decreasing rules and one can remove all rules which contain symbols of* $\mathcal{F}$ *in their left-hand side. Formally, finiteness of* $(\mathcal{P}_{\neg\mathcal{F}} \setminus \succ, \mathcal{U}_{\neg\mathcal{F}} \setminus \succ)$ *implies finiteness of* $(\mathcal{P},\mathcal{R})$.

[4] The real definition of $\mathcal{C}_\varepsilon$ in IsaFoR is slightly different due to technical reasons. Since there is no restriction on the type of variables, there might be only one variable. To this end x and y are replaced by all possible terms. And as the type of function symbols is also unrestricted there might be no fresh symbol c. However, in IsaFoR the signature is implicit where every arity is allowed. For example, the term $\mathsf{c}(\mathsf{c},\mathsf{c})$ contains one symbol $(\mathsf{c},1)$ and two symbols $(\mathsf{c},0)$ where $(\mathsf{c},1) \neq (\mathsf{c},0)$. In this way, we can always get a fresh symbol (c,n) where n is larger than all arities that occur in $\mathcal{R}$ and we use a constant $(\mathsf{d},0)$ to obtain this high arity. Hence, we define $\mathcal{C}_\varepsilon = \bigcup_{s,t}\{\mathsf{c}(s,t,\mathsf{d},\ldots,\mathsf{d}) \to s, \mathsf{c}(s,t,\mathsf{d},\ldots,\mathsf{d}) \to t\}$.

Regarding requirement (c), observe that most reduction pairs that are currently used in automated termination tools do satisfy $\mathcal{C}_\varepsilon \subseteq \succsim$. Therefore, requirement (c) of Thm. 4.1 can usually be replaced by $\mathcal{U} \subseteq \succsim$. Requirement (d) is easy to satisfy by choosing an appropriate argument filter π which depends on the reduction pair. And if $\mathcal{U}$ is chosen as the usable rules w.r.t. any known definition of usable rules, then condition (e) is satisfied. Thus, for most reduction pairs one has only replaced requirement $\mathcal{R} \subseteq \succsim$ of Thm. 4.1 by the weaker condition $\mathcal{U} \subseteq \succsim$ in Thm. 4.6.

Example 4.7. To solve the remaining DP problem $(\{(2.10)–(2.12)\}, \mathcal{R})$ of Ex. 3.2 we use a polynomial order [19] where $\mathcal{P}ol(\mathsf{set}^\sharp(x)) = \mathcal{P}ol(\mathsf{del}(y, x)) = x$, $\mathcal{P}ol(x\,{:}\,y) = \mathcal{P}ol(\mathsf{if}(\dots, y)) = y{+}1$, $\mathcal{P}ol(\mathsf{if2}^\sharp(x, y, z)) = x{+}z$, $\mathcal{P}ol(x < y) = \mathcal{P}ol(\bot) = \mathcal{P}ol(\top) = 0$, and $\mathcal{P}ol(\mathsf{c}(x, y)) = x{+}y$. A corresponding compatible argument filter π is defined by $\pi(\mathsf{set}^\sharp) = \{1\}$, $\pi(\mathsf{del}) = \pi(:) = \{2\}$, $\pi(\mathsf{if}) = \{5\}$, $\pi(\mathsf{if2}^\sharp) = \{1, 3\}$, $\pi(<) = \pi(\bot) = \pi(\top) = \emptyset$, and $\pi(\mathsf{c}) = \{1, 2\}$. For this argument filter, the minimal set of usable rules is $\mathcal{U} = \{(2.1), (2.2), (2.4)–(2.8)\}$. Note that it would also be accepted if, e.g., Rule (2.3) would be added.

Then all conditions of Thm. 4.6 are satisfied and one can remove Pair (2.10) as it is strictly decreasing. The remaining DP problem $(\{(2.11),(2.12)\}, \mathcal{R})$ is then easily solved by another application of the reduction pair processor where one chooses $\mathcal{P}ol(\mathsf{set}^\sharp(x)) = 0$, $\mathcal{P}ol(\mathsf{if2}^\sharp(x, y, z)) = 1$, $\mathcal{P}ol(\mathsf{c}(x, y)) = x + y$, and where $\mathcal{U} = \emptyset$.

In theory `UsableRules` we have proven Thm. 4.6. Although a similar proof has already been performed by the authors of [10] and [12]—on paper, not formalized—this proof has never been published in some reviewed article, it is only available in [22]. Moreover, our proof is not just a formalization of the proof in [22], but there are some essential differences which are pointed out in the following.

The standard proof of Thm. 4.6 is by transforming a minimal chain $t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \dots$ into a chain over *filtered* terms $\pi(t_1)\delta_1 \to^*_{\pi(\mathcal{U})\cup\mathcal{C}_\varepsilon} \pi(s_2)\delta_2 \to_{\pi(\mathcal{P})} \pi(t_2)\delta_2 \dots$ and then uses the preconditions of the theorem to show that certain pairs of $\pi(\mathcal{P})$ (and therefore also pairs of $\mathcal{P}$) cannot occur infinitely often. Here, one uses a transformation $\mathcal{I}_\pi$ which transforms σ_i into δ_i. For the second part of the theorem where $\succ$ is monotone, one requires another transformation $\mathcal{I}$ which does not apply any argument filter. Hence, there are two transformations $\mathcal{I}$ and $\mathcal{I}_\pi$ where for both transformations similar results are shown.

In our formalization we were able to simplify the proofs considerably by not constructing filtered terms. Moreover, we use the same transformation $\mathcal{I}$ for both parts of the theorem.

A problem in the standard proof of the reduction pair processor with usable rules is the implicit assumption that the TRS $\mathcal{R}$ meets the variable condition, i.e., $\mathcal{V}(\ell) \supseteq \mathcal{V}(r)$ and $\ell \notin \mathcal{V}$ for all rules $\ell \to r \in \mathcal{R}$. Although in practice this condition is nearly always satisfied, we have to deal with this assumption, where there are three alternatives. First, one can check that the TRS $\mathcal{R}$ meets the variable condition whenever the theorem is applied on some concrete DP problem $(\mathcal{P}, \mathcal{R})$. This would clearly increase the runtime for certifying a given proof. Second, one can define finiteness of DP problems or soundness of processors in a way that it incorporates the variable condition. However, this will make the development of other processors more complicated which do not care about the variable condition. And third, one can try to prove Thm. 4.6 without assuming the variable condition. This is the alternative we have finally formalized and which does not appear in the literature so far.

For the upcoming formal definition of $\mathcal{I}$ we essentially use a combination of the definitions of [9] and [10] where $x \# xs$ denotes the Isabelle list with head x and tail xs.

Definition 4.8. Let $\mathcal{R}$ and $\mathcal{U}$ be two TRSs, let $\mathcal{F}$ be some signature, and let c be the binary symbol of $\mathcal{C}_\varepsilon$. We define $\mathcal{I}$ as a function from terms to terms as follows:

$$\begin{aligned}
\mathsf{comb}([t]) &= t,\\
\mathsf{comb}(t \# s \# ts) &= \mathsf{c}(t, \mathsf{comb}(s \# ts)),\\
\mathsf{rewrite}(\mathcal{R}, t) &= \{C[r\sigma]_p \mid \ell \to r \in \mathcal{R}, t = C[u], \mathsf{match}(u, \ell) = \sigma\},\\
\mathcal{I}(x) &= x,\\
\mathcal{I}(f(\vec{t_n})) &= \begin{cases} f(\vec{t_n}) & \text{if } \neg\mathsf{SN}_{\mathcal{R}}(f(\vec{t_n})),\\ f(\mathcal{I}(\vec{t_n})) & \text{if } \mathsf{SN}_{\mathcal{R}}(f(\vec{t_n})),\ f \in \mathcal{F}, \text{ and } \forall \ell \to r \in \mathcal{R} \setminus \mathcal{U}.\ell \not\approx f(\mathsf{tcap}(\vec{t_n})),\\ \mathsf{comb}(f(\mathcal{I}(\vec{t_n})) \# \mathcal{I}(\mathsf{rewrite}(\mathcal{R}, f(\vec{t_n})))) & \text{otherwise.}\end{cases}
\end{aligned}$$

$\mathcal{I}$ and tcap are homeomorphically extended to operate on lists, i.e., $\mathcal{I}(\vec{t_n}) = (\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$.

The function comb just combines a non-empty list of terms into one term. It is easy to prove that one can access all terms in the list by rewriting with $\mathcal{C}_\varepsilon$: $\mathsf{comb}([\ldots, t_i, \ldots]) \to^*_{\mathcal{C}_\varepsilon} t_i$.

The function $\mathsf{rewrite}$ computes the list of one-step reducts of a given term by using a sound and complete matching algorithm match. The major difference between $\{s \mid t \to_{\mathcal{R}} s\}$ and $\mathsf{rewrite}(\mathcal{R}, t)$ is that the latter instantiates a rule by the matcher of the lhs and the corresponding redex (as usual), but it never instantiates variables which only occur in the rhs of the rule. For example, if $\mathcal{R} = \{\mathsf{a} \to x\}$ then $\{s \mid \mathsf{a} \to_{\mathcal{R}} s\}$ is the set of all terms, whereas $\mathsf{rewrite}(\mathcal{R}, \mathsf{a}) = [x]$. Hence, $\mathsf{rewrite}$ is sound ($\mathsf{rewrite}(\mathcal{R}, t) \subseteq \{s \mid t \to_{\mathcal{R}} s\}$) but in general not complete. However, under one condition completeness is achieved: whenever $t \to_{\mathcal{R}} s$ by a reduction with a rule that satisfies the variable condition, then $s \in \mathsf{rewrite}(\mathcal{R}, t)$.

The main reason for introducing $\mathsf{rewrite}$ is that without the assumption of the variable condition on $\mathcal{R}$ the set $\{s \mid t \to_{\mathcal{R}} s\}$ may be infinite. Then the definition of $\mathcal{I}$ as in [10]—where $\{\mathcal{I}(s) \mid f(\vec{t_n}) \to_{\mathcal{R}} s\}$ is used instead of $\mathcal{I}(\mathsf{rewrite}(\mathcal{R}, f(\vec{t_n})))$—does not work in combination with comb, as comb expects a list (or a *finite* set) as input. Also note that this input must be finite as one finally wants to obtain a single term containing all input terms.

The first case of $\mathcal{I}(f(\vec{t_n}))$ is mainly a technicality. Usually, $\mathcal{I}$ is only defined on terminating terms. To make $\mathcal{I}$ a total function on all terms we inserted the case distinction on $\mathsf{SN}_{\mathcal{R}}(f(\vec{t_n}))$. Termination of $\mathcal{I}$ is then proven using well-founded induction on $\xrightarrow{\rhd}_{\mathsf{SN}(\mathcal{R})}$ where in this proof the soundness result for $\mathsf{rewrite}$ is crucial.

The transformation $\mathcal{I}$ is constructed in such a way that for every reduction $t \to_{\mathcal{R}} s$ one obtains a weak decrease, provided that the usable rules and $\mathcal{C}_\varepsilon$ are weakly decreasing. Therefore, in the definition of $\mathcal{I}$ there are essentially two cases for a term $f(\vec{t_n})$. If only usable rules can be used to reduce $f(\vec{t_n})$ at the root position, then $\mathcal{I}(f(\vec{t_n}))$ is obtained by applying $\mathcal{I}$ on the arguments, resulting in $f(\mathcal{I}(\vec{t_n}))$. The corresponding reduction will then result in a weak decrease as one can also perform the reduction with the usable rules on the transformed term. The condition that only usable rules are applicable is ensured by demanding that no lhs of a non-usable rule in $\mathcal{R} \setminus \mathcal{U}$ can be unified with $f(\mathsf{tcap}(\vec{t_n}))$.

Otherwise, all rules may have been used to rewrite $f(\vec{t_n})$. Then, in addition to $f(\mathcal{I}(\vec{t_n}))$ we have to store all one-step reducts of t. This is done by encoding them in a single term using comb. Now every possible reduct can be accessed using $\mathcal{C}_\varepsilon$. And since $\mathcal{C}_\varepsilon$ is weakly decreasing we obtain a weak decrease. This is proven formally in the upcoming lemma.

Lemma 4.9 (Properties of $\mathcal{I}$)**.** *Let $(\succsim, \succ)$ be a reduction pair, let $\succsim$ be π-compatible, let $\mathcal{U} \cup \mathcal{C}_\varepsilon \subseteq \succsim$, let $\mathsf{urClosed}^\pi_{\mathcal{U},\mathcal{R}}(\mathcal{U})$, and let the rhss of $\mathcal{U}$ be terms within $\mathcal{T}(\mathcal{F},\mathcal{V})$. Let $\mathsf{SN}_\mathcal{R}(t)$, $\mathsf{SN}_\mathcal{R}(t\sigma)$, and $\mathsf{SN}_\mathcal{R}(f(\vec{t_n}))$.*

(i) *If $\mathsf{urClosed}^\pi_{\mathcal{U},\mathcal{R}}(t)$ and $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ then $t\mathcal{I}(\sigma) \succsim^* \mathcal{I}(t\sigma)$.*

(ii) *$\mathcal{I}(t\sigma) \to^*_{\mathcal{C}_\varepsilon} t\mathcal{I}(\sigma)$. And if $t \notin \mathcal{T}(\mathcal{F},\mathcal{V})$ then $\mathcal{I}(t\sigma) \to^+_{\mathcal{C}_\varepsilon} t\mathcal{I}(\sigma)$.*

(iii) *If $f(\vec{t_n}) \to_{\mathcal{R},\epsilon} s$ using a rule $\ell \to r$ and $\mathcal{I}(f(\vec{t_n})) = f(\mathcal{I}(\vec{t_n}))$ then $\ell \to r \in \mathcal{U}$, $\mathcal{I}(f(\vec{t_n})) \succsim^* \circ \to_\mathcal{U} \circ \succsim^* \mathcal{I}(s)$. And $\mathcal{I}(f(\vec{t_n})) \to^+_{\mathcal{C}_\varepsilon} \circ \to_\mathcal{U} \circ \succsim^* \mathcal{I}(s)$ if $\ell \notin \mathcal{T}(\mathcal{F},\mathcal{V})$.*

(iv) *If $f(\vec{t_n}) \to_\mathcal{R} s$ and $\mathcal{I}(f(\vec{t_n})) = \mathsf{comb}(\dots)$ then $\mathcal{I}(f(\vec{t_n})) \to^+_{\mathcal{C}_\varepsilon} \mathcal{I}(s)$.*

(v) *If $t \to_\mathcal{R} s$ then $\mathcal{I}(t) \succsim^* \mathcal{I}(s)$. Moreover, $t \to_{\mathcal{U}_{\neg\mathcal{F}}} s$ or $\mathcal{I}(t) \to^+_{\mathcal{C}_\varepsilon} \circ \succsim^* \mathcal{I}(s)$.*

(vi) *If $t \to^*_\mathcal{R} s$ then $\mathcal{I}(t) \succsim^* \mathcal{I}(s)$. Moreover, $\mathcal{I}(t) \succsim^* \circ \to_{\mathcal{C}_\varepsilon} \circ \succsim^* \mathcal{I}(s)$ or $t \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s$.*

Here, $\mathcal{I}$ is homeomorphically extended to substitutions, i.e., $\mathcal{I}(\sigma)(x) = \mathcal{I}(\sigma(x))$.

In the following proof sketch of Lem. 4.9, all essential points are included, especially those where we deviate from the standard proofs.

Proof. The proof of (vi) is a straight-forward induction on the reduction length using (v).

We prove (v), by induction over t. First note that t is not a variable. Otherwise, there would be some $x \to r \in \mathcal{R}$, contradicting $\mathsf{SN}_\mathcal{R}(t)$. Hence the base case is trivial. In the step-case, we make a case distinction on how $t = f(\vec{t_n})$ is transformed. The case $\mathcal{I}(t) = \mathsf{comb}(\dots)$ is solved by (iv). Otherwise, $\mathcal{I}(t) = f(\mathcal{I}(\vec{t_n}))$. For a root reduction we use (iii). Otherwise, $s = f(t_1, \dots, s_i, \dots, t_n)$ and $t_i \to_\mathcal{R} s_i$. Applying the induction hypothesis is easy, but some additional effort is required to prove $\mathcal{I}(s) = f(\mathcal{I}(t_1), \dots, \mathcal{I}(s_i), \dots, \mathcal{I}(t_n))$.

Proving (iv), essentially requires completeness of $\mathsf{rewrite}$. First we prove that if $f(\vec{t_n}) \to_\mathcal{R} s$ then the employed rule $\ell \to r$ must satisfy $\mathcal{V}(\ell) \supseteq \mathcal{V}(r)$, as otherwise $\mathsf{SN}_\mathcal{R}(f(\vec{t_n}))$ would not hold. Under this condition, completeness of $\mathsf{rewrite}$ states that $s \in \mathsf{rewrite}(\mathcal{R}, f(\vec{t_n}))$. The remaining proof of (iv) can be done by simple inductions using the definitions of comb and $\mathcal{C}_\varepsilon$.

To prove (iii), we first show that $\mathcal{I}(f(\vec{t_n})) = f(\mathcal{I}(\vec{t_n}))$ ensures that the employed rule $\ell \to r$ is usable. The reason is that $f(\vec{t_n}) = \ell\sigma$ implies $f(\mathsf{tcap}(\vec{t_n})) \approx \ell$ and hence, $\ell \to r \notin \mathcal{R} \setminus \mathcal{U}$ by the definition of $\mathcal{I}$. By the requirement on the rhss of $\mathcal{U}$ we know that $r \in \mathcal{T}(\mathcal{F},\mathcal{V})$. Hence, we can build the following steps using (ii) and (i) in combination with $\mathsf{urClosed}^\pi_{\mathcal{U},\mathcal{R}}(\mathcal{U})$: $\mathcal{I}(f(\vec{t_n})) = \mathcal{I}(\ell\sigma) \succsim^* \ell\mathcal{I}(\sigma) \to_\mathcal{U} r\mathcal{I}(\sigma) \succsim^* \mathcal{I}(r\sigma) = \mathcal{I}(s)$. And if $\ell \notin \mathcal{T}(\mathcal{F},\mathcal{V})$ then we additionally get $\mathcal{I}(\ell\sigma) \to^+_{\mathcal{C}_\varepsilon} \ell\mathcal{I}(\sigma)$ by (ii).

Proving (ii), is a straight-forward induction on t.

And finally, for (i), we also use induction on t. In the step-case we first prove that $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ in combination with $\mathsf{urClosed}^\pi_{\mathcal{U},\mathcal{R}}(f(\vec{t_n}))$ implies $f(\mathcal{I}(\vec{t_n}\sigma)) = \mathcal{I}(f(\vec{t_n}\sigma))$. Then, for all argument positions $i \in \pi(f)$, we apply the induction hypothesis to obtain $t_i\mathcal{I}(\sigma) \succsim^* \mathcal{I}(t_i\sigma)$. Then, by monotonicity of $\succsim$, we obtain $f(\dots, t_i\mathcal{I}(\sigma), \dots) \succsim^* f(\dots, \mathcal{I}(t_i\sigma), \dots)$. For all other positions, π-compatibility of $\succsim$ provides the same inequality. ∎

With the help of Lem. 4.9 it is now possible to prove the main result of this section.

Proof of Thm. 4.6. Assume that there is a minimal infinite chain where $s_i\sigma_i \to_\mathcal{P} t_i\sigma_i \to^*_\mathcal{R} s_{i+1}\sigma_{i+1}$ and $\mathsf{SN}_\mathcal{R}(t_i\sigma_i)$ for all i. Let $\mathcal{F}$ be the set of all symbols that occur in rhss of $\mathcal{P} \cup \mathcal{U}$. Then by the conditions of the theorem and by using Lem. 4.9 (i), (vi), and (ii), for all i we conclude

$$s_i\mathcal{I}(\sigma_i) \to_\mathcal{P} t_i\mathcal{I}(\sigma_i) \succsim^* \mathcal{I}(t_i\sigma_i) \succsim^* \mathcal{I}(s_{i+1}\sigma_{i+1}) \succsim^* s_{i+1}\mathcal{I}(\sigma_{i+1}). \tag{‡}$$

By using $\mathcal{P} \subseteq \succ \cup \succsim$ we obtain a strict or weak decrease between every two terms $s_i\mathcal{I}(\sigma_i)$ and $s_{i+1}\mathcal{I}(\sigma_{i+1})$. Thus, by Lem. 3.4, the strictly decreasing pairs can only occur finitely often. This shows that there must be some n such that for all i, $s_{i+n} \to t_{i+n} \in \mathcal{P} \setminus \succ$. Hence, there is an infinite minimal $(\mathcal{P} \setminus \succ, \mathcal{R})$-chain.

If additionally, $\mathcal{C}_\varepsilon \subseteq \succ$ and $\succ$ is monotone, we first prove that there is some n with $t_{n+i}\sigma_{n+i} \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s_{n+i+1}\sigma_{n+i+1}$ and $s_{n+i} \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ for all i. If this would not be the case, then infinitely often $t_i\sigma \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s_{i+1}\sigma_{i+1}$ does not hold or infinitely often $s_i \notin \mathcal{T}(\mathcal{F}, \mathcal{V})$. Hence, by Lem. 4.9(vi), for infinitely many i, $\mathcal{I}(t_i\sigma_i) \succsim^* \circ \to_{\mathcal{C}_\varepsilon} \circ \succsim^* \mathcal{I}(s_{i+1}\sigma_{i+1})$ or by Lem. 4.9(ii), for infinitely many i, $\mathcal{I}(s_i\sigma_i) \to^+_{\mathcal{C}_\varepsilon} s_i\mathcal{I}(\sigma_i)$. As $\succ$ contains $\mathcal{C}_\varepsilon$ and is monotone, we also have $\to_{\mathcal{C}_\varepsilon} \subseteq \succ$, and hence in both cases we obtain infinitely many strict decreases. Using the same reasoning as for (‡), we have infinitely many i with $s_i\mathcal{I}(\sigma_i)$ $(\succ \cup \succsim) \circ \succsim^* \circ \succ \circ \succsim^*$ $s_{i+1}\mathcal{I}(\sigma_{i+1})$ and for the remaining is we can use the previous results showing $s_i\mathcal{I}(\sigma)$ $(\succ \cup \succsim) \circ \succsim^*$ $s_{i+1}\mathcal{I}(\sigma_{i+1})$. Then, using Lem. 3.4 where $N = (\succ \cup \succsim)^*$ and $S = N \circ \succ \circ N$, yields a contradiction.

Hence, for all i we obtain $s_{n+i}\sigma_{n+i} \to_{\mathcal{P}_{\neg\mathcal{F}}} t_{n+1}\sigma_{n+i} \to^*_{\mathcal{U}_{\neg\mathcal{F}}} s_{n+i+1}\sigma_{n+i+1}$. Since $\mathcal{P} \cup \mathcal{U} \subseteq \succ \cup \succsim$ and since both $\succ$ and $\succsim$ are monotone and stable, we conclude (again by Lem. 3.4) that from some point onwards only rules from $(\mathcal{P}_{\neg\mathcal{F}} \cup \mathcal{U}_{\neg\mathcal{F}}) \setminus \succ$ are used. Hence, we have constructed a $(\mathcal{P}_{\neg\mathcal{F}} \setminus \succ, \mathcal{U}_{\neg\mathcal{F}} \setminus \succ)$-chain which is also minimal since $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ implies $\mathsf{SN}_{\mathcal{U}_{\neg\mathcal{F}} \setminus \succ}(t_i\sigma_i)$ as $\mathcal{U}_{\neg\mathcal{F}} \subseteq \mathcal{R}$ by the requirement $\mathcal{U} \subseteq \mathcal{R}$. ■

To obtain an executable function which checks for correct applications of Thm. 4.6 it is only demanded that the input and output DP problem, the reduction pair (without the details of the fresh symbol c), and the usable rules are given. The corresponding interpretation / precedence / ... for c is then added automatically. Moreover, the argument filter is constructed from the reduction pair. For example, for polynomial interpretations one always defines π in a way that $i \in \pi(f)$ iff x_i occurs within $\mathcal{P}ol(f(\vec{x}_n))$. In this way, $\succsim$ is always π-compatible and $\mathcal{C}_\varepsilon \subseteq \succsim$. Hence, for the automation of Thm. 4.6 where $\succ$ is not monotone, one only has to check $\mathcal{P} \subseteq \succ \cup \succsim$, $\mathcal{U} \subseteq \succsim$, and $\mathsf{urClosed}^\pi_{\mathcal{U}}(\mathcal{P}, \mathcal{R})$ since the remaining requirements are satisfied by construction.

For the other case, where also rules of $\mathcal{R}$ are deleted, it is additionally checked that $\succ$ is monotone and that $\mathcal{U} \subseteq \mathcal{R}$. To achieve the former for polynomials, it is ensured that the coefficients of all variables are larger than zero and that all remaining parts of the polynomial are non-negative. For path orders in combination with argument filters, it is ensured that no argument is dropped, i.e., the argument filter may only permute and duplicate arguments.

5. Experiments

In the end, what we want to have is a workflow which automatically certifies or rejects a given termination proof in CPF-format (a common format for termination proofs that is supported by all certifiers).[5] To this end, we have to parse the proof, detect which processors have been applied on which DP problems, and ensure that the preconditions of every processor are met. We achieved this goal by writing a CPF parser and for each processor an executable function which checks the preconditions. If a processor application cannot be certified, the function rejects, providing an informative error message. As the parser and

[5] http://cl-informatik.uibk.ac.at/software/cpf/

the check-functions are written in Isabelle, we just invoke Isabelle's code-generator [13] to obtain the executable program CeTA from IsaFoR.

This is in contrast to the other two certifiers CiME/Coccinelle and Rainbow/CoLoR.[6] Both of them provide a parser (as part of CiME and Rainbow) that takes a termination proof and produces a Coq-script as output. The resulting script refers to facts proven in Coccinelle and CoLoR, respectively, which can then be checked by Coq.

For more details on this difference and the architecture of the overall proof-checking function in CeTA we refer to [23].

To measure the impact of our results we used 5 strategies for the two termination tools AProVE [11] and TTT2 [18].

- In the ***basic*** strategy the termination tools only use the dependency graph processor and the reduction pair processor without usable rules. (These are the only techniques that have been described in [23].)
- The ***sc*** strategy is an extension of ***basic*** by the subterm criterion processor.
- Similarly, ***ur*** is like ***basic*** except that usable rules may be used.
- The fourth strategy, ***sc+ur***, is a combination of the previous three.
- Finally, ***full***, is a strategy where all CeTA-certifiable processors may be used. We refer to `http://cl-informatik.uibk.ac.at/software/ceta/versions.php` for the details where all techniques are listed. (Our experiments have been performed using CeTA version 1.10.)

Note that for ***basic***, ***sc***, ***ur***, and ***sc+ur***, we only take linear polynomial interpretations as reduction pairs, whereas in ***full*** also other reduction pairs are used.

For each of the 2132 standard TRSs in the Termination Problem Database (version 7.0.2)[7] and for each strategy, we first ran both termination tools for at most one minute and then tried to certify all successful proofs with CeTA. The experiments were performed on a machine with 8 Dual Core AMD Opteron 885 processors and 64 GB RAM running Linux. An overview of the results is given in the following table where the column labels A, C, and T, refer to AProVE, CeTA, and TTT2, respectively.

	basic		***sc***		***ur***		***sc+ur***		***full***	
	A	C	A	C	A	C	A	C	A	C
`YES`	453	453	566	566	681	681	684	684	1242	1242
avg. time		0.063		0.051		0.064		0.061		0.051
	T	C	T	C	T	C	T	C	T	C
`YES`	439	439	553	553	663	663	669	669	1223	1223
avg. time		0.059		0.048		0.065		0.062		0.074

The table rows show successful termination proofs / certificates for termination proofs (`YES`), and the average time for certifying (in seconds). All details on the experiments are available on CeTA's website.

When comparing ***basic*** with ***sc+ur*** one can observe that adding the subterm criterion and usable rules helps to increase the number of certified termination proofs by over 50% for both termination tools. Moreover, checking the additional application conditions of the new techniques—where $\mathsf{urClosed}^{\pi}_{\mathcal{U}}(\mathcal{P},\mathcal{R})$ is the most expensive one—does not have any measurable impact on the certification time. That checking AProVE's proofs is slightly

[6]Note that for a restricted set of techniques, CoLoR also features code-generation.

[7]available at `http://termcomp.uibk.ac.at/`

faster is explained by the fact that TTT2 always produces proofs with polynomial orders over the rationals, even if all coefficients are naturals. And thus, for TTT2's proofs, CeTA always has to perform computations over the rationals.

Our results also helped to win the annual termination competition for certified termination of TRSs in 2009.[8] First several termination tools were run to generate proofs on a random subset of 365 TRSs from the TPDB. For this, the tools where usually configured for a specific certifier in mind by restricting the set of termination techniques correspondingly. Then, all certifiers were run on all proofs. The following table summarizes the results, where the bold entries correspond to those proofs which were constructed specifically for that certifier.

tool	AProVE	TTT2	AProVE	CiME	AProVE	total
intended certifier	CeTA		Coccinelle		CoLoR	
proofs	259	264	165	56	220	964
CeTA	**259**	**264**	94	50	107	774
Coccinelle	12	2	**104**	**55**	30	203
CoLoR	67	53	92	9	**178**	399

We observe that many proofs generated for CeTA cannot be handled by the other certifiers—only between 1 % and 26 % of these proofs have been certified—where one major reason is that the other certifiers do not incorporate usable rules.

Looking at the other direction we see, that even if the termination tool produced proofs for another certifier, CeTA (version 1.09) achieved between 60 % and 90 % of the score of the intended certifier.

In total, only 190 proofs have been rejected by CeTAin the competition. Of these proofs, 65 are supported in the meantime (the competition version did not feature monotone matrix interpretations [7], which are supported by version 1.10), 117 contain unsupported techniques (non-linear polynomial orders and RPO [6]), 6 are obviously buggy (e.g., the subterm criterion is applied with a projection that maps a binary symbol to its third argument), and 2 are faulty (some LPO was wrongly applied and some argument filter delivers an unsolvable constraint). (At least for one of these proofs we know that this was due to an output bug of the proof producing tool.)

6. Conclusion

We have presented the first formalization of two important termination techniques within the theorem prover Isabelle/HOL: the subterm criterion and the reduction pair processor with usable rules, where we combined the improvements of [10] and [12]. The integration of these techniques into our termination proof certifier CeTA allowed to certify significantly more termination proofs.

However, there are several termination techniques that have not been certified by now. To change this, in the future we aim at certifying several techniques for innermost termination like narrowing, rewriting, and instantiation [1, 12], or estimations of innermost dependency graphs [1, 10, 14].

[8] http://termcomp.uibk.ac.at/termcomp/competition/certificationResults.seam?cat=10235&comp=101722

References

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
[3] F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *Proc. WST'06*, pages 69–73, 2006.
[4] É. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain. A3PAT, an approach for certified automated termination proofs. In *Proc. PEPM'10*. To appear.
[5] É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proc. FroCoS'07*, LNAI 4720, pages 148–162, 2007.
[6] N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3:69–116, 1987.
[7] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.
[8] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *Proc. RTA'08*, LNCS 5117, pages 110–125, 2008.
[9] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR'04*, LNAI 3452, pages 301–331, 2005.
[10] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS'05*, LNAI 3717, pages 216–231, 2005.
[11] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. In *Proc. IJCAR'06*, LNAI 4130, pages 281–286, 2006.
[12] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
[13] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Proc. FLOPS'10*. To appear.
[14] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.
[15] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
[16] S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.
[17] A. Koprowski and J. Waldmann. Arctic termination ...below zero. In *Proc. RTA'08*, LNCS 5117, pages 202–216, 2008.
[18] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. RTA'09*, volume 5595 of *LNCS*, pages 295–304, 2009.
[19] D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
[20] S. Lucas. Polynomials over the reals in proofs of termination: From theory to practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.
[21] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
[22] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen University, 2007. Available as Technical Report AIB-2007-17, `http://aib.informatik.rwth-aachen.de/2007/2007-17.pdf`.
[23] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs'09*, LNCS 5674, pages 452–468, 2009.
[24] X. Urbain. Modular & incremental automated termination proofs. *Journal of Automated Reasoning*, 32(4):315–355, 2004.
[25] H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 341-356
http://rewriting.loria.fr/rta/

TERMINATION OF LINEAR BOUNDED TERM REWRITING SYSTEMS

IRÈNE DURAND [1] AND GÉRAUD SÉNIZERGUES [1] AND MARC SYLVESTRE [1]

[1] LaBRI, Université Bordeaux 1
351 Cours de la libération, 33405 Talence cedex, France
E-mail address, I. Durand: `idurand@labri.fr`
E-mail address, G. Sénizergues: `ges@labri.fr`
E-mail address, M. Sylvestre: `sylvestr@labri.fr`

Abstract. For the whole class of linear term rewriting systems and for each integer k, we define k-bounded rewriting as a restriction of the usual notion of rewriting. We show that the k-bounded uniform termination, the k-bounded termination, the inverse k-bounded uniform, and the inverse k-bounded problems are decidable. The k-bounded class (BO(k)) is, by definition, the set of linear systems for which every derivation can be replaced by a k-bounded derivation. In general, for BO(k) systems, the uniform (respectively inverse uniform) k-bounded termination problem is not equivalent to the uniform (resp. inverse uniform) termination problem, and the k-bounded (respectively inverse k-bounded) termination problem is not equivalent to the termination (respectively inverse termination) problem. This leads us to define more restricted classes for which these problems are equivalent: the classes BOLP(k) of k-bounded systems that have the length preservation property. By definition, a system is BOLP(k) if every derivation of length n can be replaced by a k-bounded derivation of length n. We define the class BOLP of bounded systems that have the length preservation property as the union of all the BOLP(k) classes. The class BOLP contains (strictly) several already known classes of systems: the inverse left-basic semi-Thue systems, the linear growing term rewriting systems, the inverse Linear-Finite-Path-Ordering systems, the strongly bottom-up systems.

1. Introduction

General context. A Term-Rewriting System (TRS for short) $\mathcal{R}$ is said to be *terminating* on a term s when it does not admit any infinite derivation starting from s. It is said to be *inverse terminating* on s when the system $\mathcal{R}^{-1}$ terminates on s. The TRS $\mathcal{R}$ is said to be *uniformly terminating* (u-terminating for short) when it does not admit any infinite derivation, and it is said to be *inverse u-terminating* when the system $\mathcal{R}^{-1}$ u-terminates. The u-termination property is part of the definition of a *complete* TRS, which is a useful algebraic notion. These properties are also pertinent for TRSs which are models of functional programs or any kind of computational processes. It is well-known that these problems are undecidable

1998 ACM Subject Classification: Primary: F.4.2, Secondary: F.3.2, F.4.1.
Key words and phrases: term rewriting, termination, rewriting strategy.

© I. Durand, G. Sénizergues, and M. Sylvestre
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.341

for general finite TRS ([7]) and even for quite restricted subclasses of TRS (see [2],[10] for example). Nevertheless, because of its importance, many techniques have been developed in order to prove uniform-termination (u-termination for short) and termination of TRSs (see in particular [3],[9, section 1.3], [14, chap. 6]) or even to decide automatically u-termination or termination, but for specific classes of TRS.

Contents. The present paper follows the last trend of research quoted above:
1- we show that u-termination, inverse u-termination, termination, inverse termination are decidable for a particular strategy that we call bounded rewriting,
2- we deduce from this decision procedure that u-termination, inverse u-termination, termination and inverse termination problems are decidable for some classes of TRS.

We define a new rewriting strategy for linear TRSs called *bounded* rewriting. Let $k \in \mathbb{N}$. Intuitively, a derivation is said to be *k-bounded* ($\mathsf{bo}(k)$) if when a rewriting rule is applied, the parts of the substitution located at a depth greater than k are not used further in the derivation, i.e. do not match a left-handside of a rule applied further. A TRS $\mathcal{R}$ will be said to be $\mathsf{bo}(k)$ if for any derivation $s \to^*_{\mathcal{R}} t$, there exists a $\mathsf{bo}(k)$ derivation $s \;{}_{\mathsf{bo}(k)}\!\!\to^*_{\mathcal{R}} t$. The class of $\mathsf{bo}(k)$ TRSs is denoted by $\mathsf{BO}(k)$, and the class of bounded TRSs BO is $\bigcup_{k \in \mathbb{N}} \mathsf{BO}(k)$. A TRS will be said to $\mathsf{bo}(k)$-terminates on a term s if there is no infinite $\mathsf{bo}(k)$-derivation starting from s. It is said to be uniformly $\mathsf{bo}(k)$-terminating (u-$\mathsf{bo}(k)$-terminating for short) if there is no infinite $\mathsf{bo}(k)$-derivation. The main result of this paper is the decidability of the u-$\mathsf{bo}(k)$-termination problem and of the $\mathsf{bo}(k)$-termination problem. We also prove in section 6 that the inverse u-$\mathsf{bo}(k)$-termination and the inverse $\mathsf{bo}(k)$-termination problems are decidable. This rewriting strategy is closely related to the *bottom-up* strategy introduced in [4]: every bottom-up TRS is bounded, and for every bounded TRS, there is an equivalent TRS which is bottom-up. Both strategies are defined using marking tools, but the definition of the bounded strategy is simpler and more intuitive. For every linear TRS $(\mathcal{R}, \mathcal{F})$ and every integer k, there is a TRS $(\mathcal{R}', \mathcal{F})$ such that for every $s, t \in \mathcal{T}(\mathcal{F})$:

- there is a derivation of length n from s to t in $\mathcal{R}$ iff there is a derivation of length n from s to t in $\mathcal{R}'$,
- there is a $\mathsf{bo}(k)$-derivation of length n from s to t in $\mathcal{R}$ iff there is a $\mathsf{bo}(0)$-derivation of length n form s to t in $\mathcal{R}'$.

Thus, it is sufficient to prove that the u-$\mathsf{bo}(0)$-termination and the $\mathsf{bo}(0)$-termination problems are decidable to obtain the decidability of the u-$\mathsf{bo}(k)$-termination and the $\mathsf{bo}(k)$-termination problems. Following the idea developed for the bottom-up strategy, we use a ground TRS $\mathcal{S} \cup \mathcal{A}$ to simulate $\mathsf{bo}(0)$-derivations. This construction is made in such a way that the existence of an infinite $\mathsf{bo}(0)$-derivation starting from a term s in $\mathcal{R}$ is equivalent to the existence of an infinite derivation starting from s in $\mathcal{S} \cup \mathcal{A}$. It follows from the decidability of the termination and u-termination problems for ground TRS that the u-$\mathsf{bo}(0)$-termination and the $\mathsf{bo}(0)$-termination problems are decidable. The TRS $\mathcal{A}$ has rules which allow to replace any subterm of a term t located at an internal node by a leaf labeled by the constant symbol $\#$, and the TRS $\mathcal{S}$ consists of a set of rules of the form $l\sigma \to r\sigma$ where $l \to r \in \mathcal{R}$ and σ is a substitution that maps variables to an element of $\mathcal{F}_0 \cup \{\#\}$. A $\mathsf{bo}(0)$-step $C[l\sigma] \to C[r\sigma]$ in $\mathcal{R}$ is simulated in two steps : first, using $\mathcal{A}$, we reduce $C[l\sigma]$ to $C[l\sigma']$ where $l\sigma' \in \mathsf{LHS}(\mathcal{S})$, and then we apply the rule $l\sigma' \to r\sigma' \in \mathcal{S}$. We define a subclass of $\mathsf{BO}(k)$, the length preservation bottom-up class $\mathsf{BOLP}(k)$, for which:

- termination (respectively inverse termination) and k-bounded termination (resp. inverse k-bounded termination) are equivalent,

- u-termination (respectively inverse u-termination) and u-k-bounded termination (resp. inverse u-k-bounded termination) are equivalent.

A $\mathsf{BO}(k)$ TRS is $\mathsf{BOLP}(k)$ iff for every derivation $s \to_{\mathcal{R}}^{*} t$ there is a $\mathsf{bo}(k)$-derivation of same length. The class of length preservation bounded TRSs BOLP is $\bigcup_{k\in\mathbb{N}} \mathsf{BOLP}(k)$. This class contains several already known TRSs: the inverse left-basic semi-Thue systems [12], the linear growing TRS [8], the inverse Linear-Finite-Path-Overlapping TRSs [13], and the strongly bottom-up TRSs [4]. Note that a version of this article with full proofs is available at `http://dept-info.labri.fr/~sylvestr/research/papers/`.

2. Preliminaries

2.1. Words and Terms

The set $\mathbb{N}$ is the set of positive integers. A finite *word* over an alphabet A is a map $u : [0, \ell - 1] \to A$, for some $\ell \in \mathbb{N}$. The integer ℓ is the *length* of the word u and is denoted by $|u|$. The set of words over A is denoted by A^* and endowed with the usual *concatenation* operation $u, v \in A^* \mapsto u \cdot v \in A^*$. The *empty* word is denoted by ε. A word u is a prefix of a word v iff there exists some $w \in A^*$ such that $v = u \cdot w$. We denote by $u \preceq v$ the fact that u is a prefix of v. Assuming a total order on A, we denote by $\preceq_{Lex}$ the lexicographic order on words.

We assume the reader familiar with terms. We call *signature* a set $\mathcal{F}$ of symbols with fixed arity $ar : \mathcal{F} \to \mathbb{N}$. The subset of symbols of arity m is denoted by $\mathcal{F}_m$.

As usual, a set $P \subseteq \mathbb{N}^*$ is called a *tree-domain* (or, *domain*, for short) iff for every $u \in \mathbb{N}^*$,
$i \in \mathbb{N}$:

$$(u \cdot i \in P \Rightarrow u \in P) \;\&\; (u \cdot (i+1) \in P \Rightarrow u \cdot i \in P).$$

We call $P' \subseteq P$ a *subdomain* of P iff, P' is a domain and, for every $u \in P, i \in \mathbb{N}$:

$$(u \cdot i \in P' \;\&\; u \cdot (i+1) \in P) \Rightarrow u \cdot (i+1) \in P'.$$

A (first-order) *term* on a signature $\mathcal{F}$ is a partial map $t : \mathbb{N}^* \to \mathcal{F}$ whose domain is a non-empty tree-domain and which respects the arities. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of first-order terms built upon the signature $\mathcal{F} \cup \mathcal{V}$, where $\mathcal{F}$ is a finite signature and $\mathcal{V}$ is a denumerable set of variables of arity 0.

The domain of t is also called its set of *positions* and denoted by $\mathcal{P}\mathsf{os}(t)$. The set of variables of t is denoted by $\mathcal{V}\mathsf{ar}(t)$. The root symbol of t, $t(\epsilon)$ is also denoted by $\mathsf{root}(t)$. The set of variable positions (resp. non variable positions) of a term t is denoted by $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$ (resp. $\mathcal{P}\mathsf{os}_{\overline{\mathcal{V}}}(t)$). The set of *leaves* of t is the set of positions $u \in \mathcal{P}\mathsf{os}(t)$ such that $u \cdot \mathbb{N} \cap \mathcal{P}\mathsf{os}(t) = \emptyset$. It is denoted by $L\mathsf{v}(t)$. A *branch* is a set of positions P satisfying: there exists $u \in L\mathsf{v}(t)$ such that $v \in P$ iff $v \preceq u$. We write $\mathcal{P}\mathsf{os}^+(t)$ for $\mathcal{P}\mathsf{os}(t) \setminus \{\epsilon\}$. Given $v \in \mathcal{P}\mathsf{os}^+(t)$, its *father* $\mathsf{fth}(v)$ is the position u such that $v = u \cdot w$ and $|w| = 1$. Given a term t and $u \in \mathcal{P}\mathsf{os}(t)$ the *subterm of t at u* is denoted by t/u and defined by $\mathcal{P}\mathsf{os}(t/u) = \{w \mid u \cdot w \in \mathcal{P}\mathsf{os}(t)\}$ and $\forall w \in \mathcal{P}\mathsf{os}(t/u)$, $t/u(w) = t(u \cdot w)$. A term which does not contain twice the same variable is called *linear*. Given a linear term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $x \in \mathcal{V}\mathsf{ar}(t)$, we shall denote by $\mathsf{pos}(t, x)$ the position of x in t. The *depth* of a term t is inductively defined by:

- $dpt(t) := 0$ if $t \in \mathcal{V}$,
- $dpt(t) := 1$ if $t \in \mathcal{F}_0$,

- $dpt(t) := 1 + \max(\{dpt(t/i)), i \in \{0, \ldots, n-1\}\})$ if $\mathsf{root}(t) \in \mathcal{F}_n$.

A term containing no variable is called *ground*. The set of ground terms is $\mathcal{T}(\mathcal{F})$. Among all the variables, there is a special one $\square$. A term containing exactly one occurrence of $\square$ is called a *context*. A context is usually denoted as $C[]$. If v is the position of $\square$ in $C[]$, $C[t]$ denotes the term $C[]$ where t has been substituted at position v. We also denote by $C[]_v$ such a context and by $C[t]_v$ the result of the substitution. We denote by $|t| := \mathsf{Card}(\mathcal{P}\mathsf{os}(t))$ the *size* of a term t. A *substitution* σ is a mapping from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The substitution σ extends uniquely to a morphism $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, where $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$, for each $f \in \mathcal{F}$, $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Let t be a linear term and $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t) = \{u_1, \ldots, u_n\}$, where the u_i are given in lexicographic order. The term t is said to be standardized if for all $i, 1 \le i \le n$, $t/u_i = x_i$.

2.2. Term rewriting systems

A *rewrite rule* built upon the signature $\mathcal{F}$ is a pair $l \to r$ of terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We call l (resp. r) the *left-handside* (resp. *right-handside*) of the rule (*lhs* and *rhs* for short). A rule is *linear* if both its left and right-handsides are linear. A rule is *left-linear* (resp. *right-linear*) if its left-handside (resp. right-hanside) is linear. Given a set of rules $\mathcal{R}$, we denote by $\mathsf{LHS}(\mathcal{R})$ the set $\{l \mid l \to r \in \mathcal{R}\}$. A TRS is a pair $(\mathcal{R}, \mathcal{F})$ where $\mathcal{F}$ is a signature and $\mathcal{R}$ a set of rewrite rules built upon the signature $\mathcal{F}$. When $\mathcal{F}$ is clear from the context or contains exactly the symbols of $\mathcal{R}$, we may omit $\mathcal{F}$ and write simply $\mathcal{R}$. The TRS $\mathcal{R}$ is said to respect the *variable restriction* if for every $l \to r \in \mathcal{R}$, $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$. We denote by $(\mathcal{R}^{-1}, \mathcal{F})$ the TRS consisting of the rules $\{r \to l | l \to r \in \mathcal{R}\}$. Given a TRS $(\mathcal{R}, \mathcal{F})$, and two terms t_1, t_2, we say that there exists a $\mathcal{R}$*-rewriting step* between t_1 and t_2 in $\mathcal{R}$ and write $t_1 \to_{\mathcal{R}} t_2$ if there exists a context $C[]$, a rule $l \to r \in \mathcal{R}$, and a substitution σ such that $t_1 = C[l\sigma]$ and $t_2 = C[r\sigma]$. The term $l\sigma$ is called a *redex* of t_1, and $r\sigma$ is called the *contractum* of $l\sigma$. Given some $n \ge 0$, a derivation in $\mathcal{R}$ of length n from s to t is a sequence of the form $s = s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} s_n = t$. The relation $\to^n_{\mathcal{R}}$ is defined as follows: $s \to^n t$ if there exists a derivation of length n from s to t. The relation $\to^*_{\mathcal{R}}$ (resp. $\to^+_{\mathcal{R}}$) is defined by: $s \to^* t$ (resp. $s \to^+ t$) if there is some $n \ge 0$ (resp. $n > 0$) such that $s \to^n_{\mathcal{R}} t$. More generally, the notation defined in [9] will be used in proofs. A TRS is *left-linear* (resp. *right-linear*) if each of its rules is left-linear (resp. right-linear). A TRS is linear if each of its rules is linear. A TRS $\mathcal{R}$ is *growing* [8] if for every rule $l \to r \in \mathcal{R}$, and every occurrence of a variable in $\mathcal{V}\mathsf{ar}(l) \cap \mathcal{V}\mathsf{ar}(r)$, this occurence has depth 0 or 1 in l. Two TRSs $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F})$ are said to be *equivalent* if for all $n \ge 0$, $\to^n_{\mathcal{R}} = \to^n_{\mathcal{R}'}$.

3. Bounded rewriting

From now on, until the end of this paper, we suppose that all the TRSs satisfy the variable restriction. In order to define *bounded rewriting* for linear TRS, we need some marking tools. In the following we assume that $\mathcal{F}$ is a signature. We shall illustrate many of our definitions with the following TRS

Example 3.1. $\mathcal{F} = \{\mathsf{a}, \mathsf{b}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{i}\}$, $\mathcal{R}_1 = \{\mathsf{f}(x) \to \mathsf{g}(x), \mathsf{g}(\mathsf{h}(x)) \to \mathsf{i}(x), \mathsf{i}(x) \to \mathsf{a}, \mathsf{a} \to \mathsf{b}\}$.

3.1. Marking

We mark the symbols of a term using natural integers.

3.1.1. *Marked symbols.*

Definition 3.2. We define the (infinite) *signature of marked symbols*: $\mathcal{F}^{\mathbb{N}} := \{f^i \mid f \in \mathcal{F}, i \in \mathbb{N}\}$.
For $j \in \mathbb{N}$, we denote by $\mathcal{F}^{\leq j}$ the signature: $\mathcal{F}^{\leq j} := \{f^i \mid f \in \mathcal{F}, i \leq j\}$. The mapping $\mathsf{m} : \mathcal{F}^{\mathbb{N}} \to \mathbb{N}$ maps every marked symbol to its mark: $\mathsf{m}(f^i) = i$.

3.1.2. *Marked terms.*

Definition 3.3. The terms in $\mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ are called *marked* terms.

The mapping m is extended to marked terms by: if $t \in \mathcal{V}, \mathsf{m}(t) := 0$, otherwise, $\mathsf{m}(t) := \mathsf{m}(\mathsf{root}(t))$. For every $f \in \mathcal{F}$, we identify f^0 and f; it follows that $\mathcal{F} \subset \mathcal{F}^{\mathbb{N}}$, $\mathcal{T}(\mathcal{F}) \subset \mathcal{T}(\mathcal{F}^{\mathbb{N}})$ and $\mathcal{T}(\mathcal{F}, \mathcal{V}) \subset \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$. We use $\mathsf{mmax}(t)$ to denote the maximal mark of a marked term t:
$\mathsf{mmax}(t) := \max\{\mathsf{m}(t/u) \mid u \in \mathcal{P}\mathsf{os}(t)\}$.

Example 3.4. $\mathsf{m}(\mathsf{a}^1) = 1, \mathsf{m}(\mathsf{i}^0(\mathsf{a}^2)) = 0, \mathsf{m}(\mathsf{h}^3(\mathsf{a}^0)) = 3, \mathsf{m}(\mathsf{h}^1(x)) = 1, \mathsf{m}(x) = 0$, $\mathsf{mmax}(\mathsf{i}^0(\mathsf{a}^1)) = 1, \mathsf{mmax}(x) = 0$.

Definition 3.5. Given $t \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ and $i \in \mathbb{N}$, we define the marked term t^i whose marks are all equal to i:

if t is a variable x	$t^i := x$
if t is a constant c	$t^i := c^i$
otherwise $t = f(t_1, \ldots, t_n)$, where $n \geq 1$	$t^i := f^i({t_1}^i, \ldots, {t_n}^i)$

This marking extends to sets of terms S ($S^i := \{t^i \mid t \in S\}$) and substitutions σ ($\sigma^i : x \mapsto (x\sigma)^i$).
Notation: in the sequel, given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $\bar{t}$ will always refer to a term of $\mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ such that $\bar{t}^0 = t$.

Definition 3.6. For every marked term $\bar{t}$, we denote by $\widehat{\bar{t}}$ the unique marked term such that:

$$\widehat{\bar{t}}^0 := \bar{t}^0, \ \ \forall u \in \mathcal{P}\mathsf{os}_{\overline{\mathcal{V}}}(t), \mathsf{m}(\widehat{\bar{t}}/u) := \max(\mathsf{m}(\bar{t}/u), |u| + 1).$$

We extend this definition to marked substitutions ($\widehat{\overline{\sigma}} : x \mapsto \widehat{x\overline{\sigma}}$) and sets of terms ($\widehat{\overline{S}} := \{\widehat{\overline{s}} \mid \overline{s} \in \overline{S}\}$).

Example 3.7. Let $\overline{t_1} = \mathsf{f}^0(\mathsf{f}^1(x))$, and $\overline{t_2} = \mathsf{f}^2(\mathsf{f}^2(\mathsf{h}^2(\mathsf{a}^2)))$. We have: $\widehat{\overline{t_1}} = \mathsf{f}^1(\mathsf{f}^2(x))$, $\widehat{\overline{t_2}} = \mathsf{f}^2(\mathsf{f}^2(\mathsf{h}^3(\mathsf{a}^4)))$.

3.2. Marked rewriting

Let $\mathcal{R}$ be a linear TRS, and let $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$. Let us suppose that $\overline{s}$ decomposes as

$$\overline{s} = \overline{C}[\bar{l}\overline{\sigma}]_v, \quad \text{with} \quad (l, r) \in \mathcal{R}, \tag{3.1}$$

for some marked context $\overline{C}[]_v$ and substitution $\overline{\sigma}$. We then write $\overline{s} \circ\!\!\to \bar{t}$ when

$$\overline{s} = \overline{C}[\bar{l}\overline{\sigma}], \;\; \bar{t} = \overline{C}[r\widehat{\overline{\sigma}}]. \tag{3.2}$$

More precisely, an ordered pair of marked terms $(\overline{s}, \bar{t})$ is linked by the relation $\circ\!\!\to$ iff, there exists $\overline{C}[]_v, (l, r), \bar{l}, \overline{\sigma}$ fulfilling equations (3.1-3.2).

The map $\overline{s} \mapsto \overline{s}^0$ (from marked terms to unmarked terms) extends into a map from marked derivations to unmarked derivations: every

$$\overline{s_0} = \overline{C_0}[\overline{l_0}\overline{\sigma_0}]_{v_0} \circ\!\!\to \overline{C_0}[r_0\widehat{\overline{\sigma_0}}]_{v_0} = \overline{s_1} \circ\!\!\to \ldots \circ\!\!\to \overline{C_{n-1}}[r_{n-1}\widehat{\overline{\sigma_{n-1}}}]_{v_{n-1}} = \overline{s_n} \tag{3.3}$$

is mapped to the derivation

$$s_0 = C_0[l_0\sigma_0]_{v_0} \to C_0[r_0\sigma_0]_{v_0} = s_1 \to \ldots \to C_{n-1}[r_{n-1}\sigma_{n-1}]_{v_{n-1}} = s_n. \tag{3.4}$$

The context $\overline{C_i}[]_{v_i}$, the rule (l_i, r_i), the marked version $\bar{l_i}$ of l_i and the substitution $\overline{\sigma_i}$ completely determine $\overline{s_{i+1}}$. Thus, for every fixed pair $(\overline{s_0}, s_0)$, this map is a bijection from the set of derivations (3.3), to the set of derivations (3.4).

From now on, each time we deal with a derivation $s \to^* t$ between two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we may implicitly decompose it as (3.4) where n is the length of the derivation, $s = s_0$ and $t = s_n$.

3.3. Bounded derivations

Definition 3.8. The marked derivation (3.3) is *k-bounded* ($\mathsf{bo}(k)$) if the following assertions hold for every $0 \leq i < n$:

- if $l_i \notin \mathcal{V}$, $\mathsf{mmax}(\overline{l_i}) \leq k$,
- if $l_i \in \mathcal{V}$, $\sup(\{\mathsf{m}(\overline{s_i}/u) | u \prec v_i\}) \leq k$.

The derivation (3.4) is $\mathsf{bo}(k)$ if the corresponding marked derivation (3.3) is $\mathsf{bo}(k)$.

Example 3.9. Let us consider the following derivations in $\mathcal{R}_1$:

(1) $\mathsf{f(h(a))} \to \mathsf{g(h(a))} \to \mathsf{i(a)} \to \mathsf{a}$
(2) $\mathsf{f(h(a))} \to \mathsf{g(h(a))} \to \mathsf{g(h(b))} \to \mathsf{i(b)} \to \mathsf{a}$

The first derivation is $\mathsf{bo}(1)$ since the associated marked derivation is $\mathsf{bo}(1)$:
$\mathsf{f(h(a))} \circ\!\!\to \mathsf{g(h^1(a^2))} \circ\!\!\to \mathsf{i(a^2)} \circ\!\!\to \mathsf{a}$. The second one is $\mathsf{bo}(2)$:
$\mathsf{f(h(a))} \circ\!\!\to \mathsf{g(h^1(a^2))} \circ\!\!\to \mathsf{g(h^1(b))} \circ\!\!\to \mathsf{i(b^1)} \circ\!\!\to \mathsf{a}$.

Let $k \in \mathbb{N}$. It is clear that the composition of two $\mathsf{bo}(k)$ marked derivations is $\mathsf{bo}(k)$ too, but the composition of two unmarked $\mathsf{bo}(k)$-derivations might not be $\mathsf{bo}(k)$, as shown in the following example:

Example 3.10. The two derivations in $\mathcal{R}_1$: $\mathsf{f(h(a))} \to \mathsf{g(h(a))}$ and $\mathsf{g(h(a))} \to \mathsf{i(a)} \to \mathsf{a}$ are $\mathsf{bo}(0)$ while the derivation: $\mathsf{f(h(a))} \to \mathsf{g(h(a))} \to \mathsf{i(a)} \to \mathsf{a}$ is not $\mathsf{bo}(0)$ (but is $\mathsf{bu}(1)$).

In the following we thus (mainly) manipulate *marked* $\mathsf{bo}(k)$-derivations. Let us introduce some convenient notations.

Definition 3.11. Let $n, k \in \mathbb{N}$. The binary relation ${}_{\mathsf{bo}(k)}\circ\!\!\rightarrow^n_{\mathcal{R}}$ over $\mathcal{T}(\mathcal{F}^{\mathbb{N}})$ is defined by: $\bar{s}\ {}_{\mathsf{bo}(k)}\circ\!\!\rightarrow^n_{\mathcal{R}}\ \bar{t}$ iff there exists a $\mathsf{bo}(k)$-marked derivation from $\bar{s}$ to $\bar{t}$ of length n. The binary relation ${}_{\mathsf{bo}(k)}\circ\!\!\rightarrow^*_{\mathcal{R}}$ is defined by: $\bar{s}\ {}_{\mathsf{bo}(k)}\circ\!\!\rightarrow^*_{\mathcal{R}}\ \bar{t}$ iff there exists $m \in \mathbb{N}$ such that $\bar{s}\ {}_{\mathsf{bo}(k)}\circ\!\!\rightarrow^m_{\mathcal{R}}\ \bar{t}$. The binary relation ${}_{\mathsf{bo}(k)}\!\!\rightarrow^n_{\mathcal{R}}$ over $\mathcal{T}(\mathcal{F})$ is defined by: $s\ {}_{\mathsf{bo}(k)}\!\!\rightarrow^n_{\mathcal{R}}\ t$ iff there exists a $\mathsf{bo}(k)$-derivation from s to t of length n. The binary relation ${}_{\mathsf{bo}(k)}\!\!\rightarrow^*_{\mathcal{R}}$ is defined by: $s\ {}_{\mathsf{bo}(k)}\!\!\rightarrow^*_{\mathcal{R}}\ t$ iff there exists $m \in \mathbb{N}$ such that $s\ {}_{\mathsf{bo}(k)}\!\!\rightarrow^m_{\mathcal{R}}\ t$.

Next lemma shows that the study of $\mathsf{bo}(k)$-derivations can be reduced to the study of $\mathsf{bo}(0)$-derivations.

Lemma 3.12. *Let $\mathcal{R}$ be a linear TRS and let $k > 0$. There exists an equivalent linear TRS $\mathcal{R}'$ such that: for all $n \in \mathbb{N}$, ${}_{\mathsf{bo}(k)}\!\!\rightarrow^n_{\mathcal{R}} = {}_{\mathsf{bo}(0)}\!\!\rightarrow^n_{\mathcal{R}'}$.*

Sketch of proof. Let $\mathcal{R}'$ be the TRS consisting of the rules:

$$\{l\sigma \rightarrow r\sigma \mid l \rightarrow r \in \mathcal{R}, \sigma : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{V}), l\sigma \text{ is standardized}, \forall x \in \mathcal{V}, dpt(x\sigma) \leq k\}.$$

One can easily check that $\mathcal{R}'$ is finite, equivalent to $\mathcal{R}$ and that, for all $n \in \mathbb{N}$, ${}_{\mathsf{bo}(k)}\!\!\rightarrow^n_{\mathcal{R}} = {}_{\mathsf{bo}(0)}\!\!\rightarrow^n_{\mathcal{R}'}$. ∎

Example 3.13. Let us consider the $\mathsf{bo}(1)$-derivation in example 3.9 $\mathsf{f}(\mathsf{h}(\mathsf{a})) \rightarrow \mathsf{g}(\mathsf{h}(\mathsf{a})) \rightarrow \mathsf{i}(\mathsf{a}) \rightarrow \mathsf{a}$ and the TRS $\mathcal{R}'$ built for $\mathcal{R}_1$ and $k = 1$. We have:

$$\begin{aligned}\mathcal{R}' =\{&\mathsf{f}(x_1) \rightarrow \mathsf{g}(x_1), \mathsf{f}(\mathsf{f}(x_1)) \rightarrow \mathsf{g}(\mathsf{f}(x_1)), \mathsf{f}(\mathsf{g}(x_1)) \rightarrow \mathsf{g}(\mathsf{g}(x_1)),\\ &\mathsf{f}(\mathsf{h}(x_1)) \rightarrow \mathsf{g}(\mathsf{h}(x_1)), \mathsf{f}(\mathsf{i}(x_1)) \rightarrow \mathsf{g}(\mathsf{i}(x_1)), \mathsf{f}(\mathsf{a}) \rightarrow \mathsf{g}(\mathsf{a}), \mathsf{f}(\mathsf{b}) \rightarrow \mathsf{g}(\mathsf{b}),\\ &\mathsf{g}(\mathsf{h}(x_1)) \rightarrow \mathsf{i}(x_1), \mathsf{g}(\mathsf{h}(\mathsf{f}(x_1))) \rightarrow \mathsf{i}(\mathsf{f}(x_1)), \mathsf{g}(\mathsf{h}(\mathsf{g}(x_1))) \rightarrow \mathsf{i}(\mathsf{g}(x_1)),\\ &\mathsf{g}(\mathsf{h}(\mathsf{h}(x_1))) \rightarrow \mathsf{i}(\mathsf{h}(x_1)), \mathsf{g}(\mathsf{h}(\mathsf{i}(x_1))) \rightarrow \mathsf{i}(\mathsf{i}(x_1)), \mathsf{g}(\mathsf{h}(\mathsf{a})) \rightarrow \mathsf{i}(\mathsf{a}),\\ &\mathsf{g}(\mathsf{h}(\mathsf{b})) \rightarrow \mathsf{i}(\mathsf{b}), \mathsf{i}(x_1) \rightarrow \mathsf{a}, \mathsf{i}(\mathsf{f}(x_1)) \rightarrow \mathsf{a}, \mathsf{i}(\mathsf{g}(x_1)) \rightarrow \mathsf{a},\\ &\mathsf{i}(\mathsf{h}(x_1)) \rightarrow \mathsf{a}, \mathsf{i}(\mathsf{i}(x_1)) \rightarrow \mathsf{a}, \mathsf{i}(\mathsf{a}) \rightarrow \mathsf{a}, \mathsf{i}(\mathsf{b}) \rightarrow \mathsf{a}, \mathsf{a} \rightarrow \mathsf{b}\}\end{aligned}$$

and the following $\mathsf{bo}(0)$-derivation in $\mathcal{R}'$:

$$\mathsf{f}(\mathsf{h}(\mathsf{a}))\ \circ\!\!\rightarrow_{\mathsf{f}(\mathsf{h}(x_1))\rightarrow\mathsf{g}(\mathsf{h}(x_1))}\ \mathsf{g}(\mathsf{h}(\mathsf{a}^1))\ \circ\!\!\rightarrow_{\mathsf{h}(x_1)\rightarrow\mathsf{i}(x_1)}\ \mathsf{i}(\mathsf{a}^1)\ \circ\!\!\rightarrow_{\mathsf{i}(x_1)\rightarrow\mathsf{a}}\ \mathsf{a}.$$

3.4. Bounded systems

We introduce here a hierarchy of classes of linear TRSs, based on their ability to meet the bounded restriction over derivations.

Definition 3.14. Let p be some property of derivations. A TRS $(\mathcal{R}, \mathcal{F})$ is called P if $\forall s, t \in \mathcal{T}(\mathcal{F})$ such that $s \rightarrow^*_{\mathcal{R}} t$ there exists a p-derivation from s to t.

We denote by $\mathsf{BO}(k)$ the class of $\mathsf{BO}(k)$ TRSs. One can check that, for every $k > 0$, $\mathsf{BO}(k-1) \subsetneq \mathsf{BO}(k)$. Finally, the class of *bounded systems* BO is defined by: $\mathsf{BO} = \bigcup_{k \in \mathbb{N}} \mathsf{BO}(k)$.

The class BO contains several already known classes of TRS (see section 7.4).

Remark 3.15. The most obvious extension of the BO definition to left-linear TRSs (keeping the marking process and the definitions unchanged) is not really interesting since even the TRS consisting of the rules $\{\mathsf{f}(x) \to \mathsf{g}(x,x), \mathsf{a} \to \mathsf{b}\}$ is not in BO: for every $k \in \mathbb{N}$ there is a $\mathsf{bo}(k+1)$-derivation:

$$\mathsf{f}(\mathsf{f}(\ldots \mathsf{f}(\mathsf{a})\ldots)) \to \mathsf{g}(\mathsf{f}^1(\mathsf{f}^2(\ldots(\mathsf{f}^k(\mathsf{a}^{k+1})\ldots)), \mathsf{f}^1(\mathsf{f}^2(\ldots(\mathsf{f}^k(\mathsf{a}^{k+1})\ldots)))$$
$$\to \mathsf{g}(\mathsf{f}^1(\mathsf{f}^2(\ldots(\mathsf{f}^k(\mathsf{a}^{k+1})\ldots)), \mathsf{f}^1(\mathsf{f}^2(\ldots(\mathsf{f}^k(\mathsf{b})\ldots)))$$

but there derivation from $\mathsf{f}(\mathsf{f}(\ldots \mathsf{f}(\mathsf{a})\ldots))$ to $\mathsf{g}(\mathsf{f}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{a})\ldots)), \mathsf{f}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{b})\ldots)))$ that is $\mathsf{bo}(k)$.

Let $\mathcal{R}$ be some linear TRS over the signature $\mathcal{F}$. Let us introduce a new unary symbol $f_1 \notin \mathcal{F}$ and consider the signature $\mathcal{F}_1 = \mathcal{F} \cup \{f_1\}$. We then define

$$\mathcal{R}_{f_1} = \{l \to r \in \mathcal{R}, l \notin \mathcal{V}\} \cup \{C[l] \circ\!\!\to C(r) | l \to r \in \mathcal{R}, dpt(C[]) = 1\}.$$

It is clear that $\mathcal{R}_{f_1}$ is a linear finite TRS over $\mathcal{F}_1$ such that $\mathsf{LHS}(\mathcal{R}_{f_1}) \cap \mathcal{V} = \emptyset$.

Lemma 3.16. *For every $s, t \in \mathcal{T}(\mathcal{F})$, and integers $k \geq 0$, $n \geq 0$,*

(1) $s \to^n_{\mathcal{R}} t \Leftrightarrow f_1(s) \to^n_{\mathcal{R}_{f_1}} f_1(t)$
(2) $s \;_{\mathsf{bo}(k)}\!\!\to^n_{\mathcal{R}} t \Leftrightarrow f_1(s) \;_{\mathsf{bo}(k)}\!\!\to^n_{\mathcal{R}_{f_1}} f_1(t)$ ∎

Definition 3.17. An infinite inverse $\mathsf{bo}(k)$-derivation is a derivation $s_0 \to_{\mathcal{R}^{-1}} s_1 \to_{\mathcal{R}^{-1}} \ldots \to_{\mathcal{R}^{-1}} s_n \ldots$ such that there exist $(\overline{s_m})_{\mathsf{m}} \in \mathbb{N}$, that satisfying: for all $i \in \mathbb{N}$, $\overline{s_{i+1}} \;_{\mathsf{bo}(k)}\!\circ\!\!\to_{\mathcal{R}} \overline{s_i}$.

Definition 3.18. We say that the TRS $\mathcal{R}$ $\mathsf{bo}(k)$-*terminates* (respectively *inverse* $\mathsf{bo}(k)$-*terminates*) on a term s iff there is no infinite $\mathsf{bo}(k)$-derivation (resp. inverse $\mathsf{bo}(k)$-derivation) starting from s in $\mathcal{R}$.
The $\mathsf{bo}(k)$-termination (resp. inverse $\mathsf{bo}(k)$) problem for a linear TRS $\mathcal{R}$ is the following problem:
INSTANCE: A linear TRS $\mathcal{R}$, an integer k, and a term s.
QUESTION: Does $\mathcal{R}$ $\mathsf{bo}(k)$-terminate (reps. inverse $\mathsf{bo}(k)$-terminate) on s ?

Definition 3.19. We say that the TRS $\mathcal{R}$ *u*-$\mathsf{bo}(k)$-*terminates* (respectively *inverse u*-$\mathsf{bo}(k)$-*terminates*) iff there is no infinite $\mathsf{bo}(k)$-derivation (resp. inverse $\mathsf{bo}(k)$-derivation) in $\mathcal{R}$.
The u-$\mathsf{bo}(k)$-termination (resp. inverse u-$\mathsf{bo}(k)$) problem for a linear TRS $\mathcal{R}$ is the following problem:
INSTANCE: A linear TRS $\mathcal{R}$, and an integer k.
QUESTION: Does $\mathcal{R}$ u-$\mathsf{bo}(k)$-terminate ?

The main result of this paper is the decidability of the u-$\mathsf{bo}(k)$-termination, $\mathsf{bo}(k)$-termination, inverse u-$\mathsf{bo}(k)$-termination, and inverse $\mathsf{bo}(k)$-termination problems. By lemma 3.16, it is sufficient to prove these results for TRSs $\mathcal{R}$ satisfying $\mathsf{LHS}(\mathcal{R}) \cap \mathcal{V} = \emptyset$. So, from now on until the end of section 6, we suppose that all the TRSs are satisfying this condition.

4. Simulation of bounded derivations by a ground rewriting system

In this section, we prove that a $\mathsf{bo}(0)$-derivation can be simulated using a ground TRS.

Definition 4.1. Let $\#$ be a constant such that $\# \notin \mathcal{F}_0$. Let $\mathcal{A}$ be the (infinite) TRS on $\mathcal{T}((\mathcal{F} \cup \{\#\}^{\mathbb{N}}))$ consisting of the rules:

$$\{f^i(\overline{c_1}, \dots, \overline{c_n}) \to \#^i \mid i \in \mathbb{N}, f \in \mathcal{F}_n, n > 0, \overline{c_1}, \dots, \overline{c_n} \in (\mathcal{F}_0 \cup \{\#\})^{\mathbb{N}}\}.$$

For $j \in \mathbb{N}$, we denote by $\mathcal{A}^{\leq j}$ the restriction of $\mathcal{A}$ on $\mathcal{T}((\mathcal{F} \cup \{\#\})^{\leq j})$ consisting of the rules:

$$\{f^i(\overline{c_1}, \dots, \overline{c_n}) \to \#^i \mid i \leq j, f \in \mathcal{F}_n, n > 0, \overline{c_1}, \dots, \overline{c_n} \in (\mathcal{F}_0 \cup \{\#\})^{\leq j}\}.$$

Lemma 4.2. *Let $\overline{s}, \overline{t} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\mathbb{N}})$. If $\overline{s} \to_{\mathcal{A}}^* \overline{t}$, then $\widehat{\overline{s}} \to_{\mathcal{A}}^* \widehat{\overline{t}}$.* ∎

Definition 4.3. A marked term $\overline{t} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\mathbb{N}}, \mathcal{V})$ is said to be *smoothly-increasing* (s-increasing for short) iff for every branch b, the sequence of marks on b has the form:

$$0, 0, \dots, 0, 1, 2, \dots, \ell$$

i.e. more formally: for every $w \in L\mathsf{v}(t)$, there exists some $u \preceq w$ such that,

- $\forall v \prec u, \mathsf{m}(\overline{t}/v) = 0$,
- $\mathsf{m}(\overline{t}/u) \in \{0, 1\}$,
- $\forall v \succeq u, \forall i \in \mathbb{N}$, if $v \cdot i \preceq w$ then $\mathsf{m}(\overline{t}/v \cdot i) = \mathsf{m}(\overline{t}/v) + 1$.

A substitution $\overline{\sigma}$ is said to be s-increasing if for every $x \in \mathcal{V}$, the term $x\overline{\sigma}$ is s-increasing.

Note that by definition of a s-increasing term $\overline{t}$, and since the variables are all marked by 0, for all positions $u \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$, for all $v \preceq u$, $\mathsf{m}(\overline{t}/v) = 0$.

Example 4.4. The terms $\mathsf{f}^0(\mathsf{h}^1(x))$ and $\mathsf{f}^2(\mathsf{h}^1(\mathsf{a}^2))$ are not s-increasing. The terms $\mathsf{f}^0(\mathsf{f}^0(\mathsf{h}^1(\mathsf{a}^2)))$ and $\mathsf{f}^1(\mathsf{a}^2)$ are s-increasing.

Lemma 4.5. *Let $\overline{C}[]$ and $\overline{t}$ be s-increasing. The term $\overline{C}[\overline{t}]$ is s-increasing.* ∎

Lemma 4.6. *Let $\overline{s}$ be a s-increasing term and $\overline{s}$ $_{\mathsf{bo}(0)} \circ\!\!\to_{\mathcal{R}}^* \overline{t}$. The marked term $\overline{t}$ is s-increasing.* ∎

Definition 4.7. Let $\overline{t} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\mathbb{N}}, \mathcal{V})$ be a marked term and P be a subdomain of $\mathcal{P}\mathsf{os}(t)$ such that $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t) \subseteq P$. We define $\mathsf{Red}(\overline{t}, P)$ as the unique term such that $\mathcal{P}\mathsf{os}(\mathsf{Red}(\overline{t}, P)) = P$ and such that $\overline{t} \to_{\mathcal{A}}^* \mathsf{Red}(\overline{t}, P)$.

The term $\mathsf{Red}(\overline{t}, P)$ is obtained from $\overline{t}$ by substituting the subtree $\overline{t}/u$ by the symbol $\#^{\mathsf{m}(\overline{t}/u)}$, for every position $u \in P \backslash L\mathsf{v}(t)$ such that $\forall i \in \mathbb{N}, u \cdot i \notin P$.

Lemma 4.8. *Let $\overline{t} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\mathbb{N}}, \mathcal{V})$ and P be a subdomain of $\mathcal{P}\mathsf{os}(t)$ such that $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t) \subseteq P$. We have $\mathsf{Red}(\widehat{\overline{t}}, P) = \widehat{\mathsf{Red}(\overline{t}, P)}$.* ∎

4.1. Top of a term

Definition 4.9 (Top domain of a term)**.** Let $\overline{t}$ be a s-increasing term. We define the top domain of $\overline{t}$, denoted by $\mathsf{Topd}(\overline{t})$ as: $u \in \mathsf{Topd}(\overline{t})$ iff $u \in \mathcal{P}\mathsf{os}(\overline{t}) \wedge \mathsf{m}(\overline{t}/u) \leq 1$.

Note that by definition of a s-increasing term, $\mathsf{Topd}(\overline{t})$ is a subdomain of t and since for every $u \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$, $\mathsf{m}(\overline{t}/u) = 0$, we have $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t) \subseteq \mathsf{Topd}(\overline{t})$.

Definition 4.10 (Top of a term)**.** Let $\overline{t}$ be a s-increasing term. We denote by $\mathsf{Top}(\overline{t})$ the term $\mathsf{Red}(\overline{t}, \mathsf{Topd}(\overline{t}))$.

Example 4.11. Let $\overline{t_1} = \mathsf{f}^0(\mathsf{h}^1(\mathsf{a}^2))$, $\overline{t_2} = \mathsf{f}^0(\mathsf{h}^0(\mathsf{a}^1))$. We have: $\mathsf{Topd}(\overline{t_1}) = \{\epsilon, 0\}$, $\mathsf{Topd}(\overline{t_2}) = \mathcal{P}\mathsf{os}(t_2), \mathsf{Top}(\overline{t_1}) = \mathsf{f}(\#^1), \mathsf{Top}(\overline{t_2}) = \overline{t_2}$.

Intuitively, the top of a term $\bar{t}$ will be the only part of $\bar{t}$ which could be used in a $\mathsf{bo}(0)$-derivation starting from $\bar{t}$. We extend this definition to sets of s-increasing terms ($\mathsf{Top}(\overline{S}) := \{\mathsf{Top}(\bar{t}) \mid \bar{t} \in \overline{S}\}$) and to s-increasing marked substitutions ($\mathsf{Top}(\overline{\sigma}) : x \mapsto \mathsf{Top}(x\overline{\sigma})$).

Lemma 4.12. *Let $\overline{C}[]_v, \overline{t_1}$ be s-increasing and let $\bar{t} = \overline{C}[\overline{t_1}]_v$. We have:*

$$\mathsf{Top}(\bar{t}) = \mathsf{Top}(\overline{C}[]_v)[\mathsf{Top}(\overline{t_1})]_v.$$

■

Lemma 4.13. *Let $\bar{t}$ and $\overline{\sigma}$ be s-increasing. We have:* $\mathsf{Top}(\bar{t}\overline{\sigma}) = \mathsf{Top}(\bar{t})\mathsf{Top}(\overline{\sigma})$. ■

4.2. The ground system $\mathcal{S}$

Definition 4.14. For a linear TRS $\mathcal{R}$, we consider the following ground TRS $\mathcal{S}$ over $\mathcal{T}((\mathcal{F} \cup \{\#\})^{\leq 1})$ consisting of all the rules of the form: $l\overline{\sigma} \to r\widehat{\overline{\sigma}}$, where $l \to r$ is a rule of $\mathcal{R}$, and $\overline{\sigma} : \mathcal{V} \to (\mathcal{F}_0 \cup \{\#\})^{\leq 1}$.

Note that since $\overline{\sigma} : \mathcal{V} \to (\mathcal{F}_0 \cup \{\#\})^{\leq 1}$, by definition of $\widehat{\ }$, $\widehat{\overline{\sigma}} : \mathcal{V} \to (\mathcal{F}_0 \cup \{\#\})^{\leq 1}$. The TRS $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ will be used to simulate the $\mathsf{bo}(0)$-derivations in $\mathcal{R}$.

4.3. Lifting lemma

Lemma 4.15. *Let $\overline{s'} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\mathbb{N}})$, $\bar{s}, \bar{t} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\leq 1})$. Assume that $\overline{s'} \to^*_{\mathcal{A}} \bar{s} \to_{\mathcal{S}} \bar{t}$. There exists a term $\overline{t'} \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\mathbb{N}})$ such that $\overline{s'} \ {}_{\mathsf{bo}(0)}\!\!\circ\!\!\!\to_{\mathcal{R}} \overline{t'} \to^*_{\mathcal{A}} \bar{t}$.*

Proof. We have $\bar{s} \to_{\mathcal{S}} \bar{t}$. This means that $\bar{s} = \overline{C}[l\overline{\sigma}]_v$, $\bar{t} = \overline{C}[r\widehat{\overline{\sigma}}]_v$, for some rule $l \to r \in \mathcal{R}$, marked context $\overline{C}[]_v$, and marked substitution $\overline{\sigma} : \mathcal{V} \to (\mathcal{F}_0 \cup \{\#\})^{\leq 1}$. Since $\overline{s'} \to^*_{\mathcal{A}} \bar{s}$, and since $\mathcal{A}$ goes from bottom to top, there exists a context $\overline{C}'[]_v$, a substitution $\overline{\sigma'}$ such that $\overline{s'}$ is of the form $\overline{s'} = \overline{C'}[l\overline{\sigma'}]_v$, with $\overline{C}'[]_v$ such that $\overline{C}'[] \to^*_{\mathcal{A}} \overline{C}[]$, and $\overline{\sigma'}$ such that for every $x \in \mathcal{V}\mathsf{ar}(l)$, $x\overline{\sigma'} \to^*_{\mathcal{A}} x\overline{\sigma}$. By definition of ${}_{\mathsf{bo}(0)}\!\!\circ\!\!\!\to$, $\overline{s'} = \overline{C'}[l\overline{\sigma'}]_v \ {}_{\mathsf{bo}(0)}\!\!\circ\!\!\!\to_{\mathcal{R}} \overline{t'} = \overline{C}'[r\widehat{\overline{\sigma'}}]_v$. By Lemma 4.2, for every $x \in \mathcal{V}\mathsf{ar}(r)$, $x\widehat{\overline{\sigma'}} \to^*_{\mathcal{A}} x\widehat{\overline{\sigma}}$. Hence, $\overline{t'} = \overline{C'}[r\widehat{\overline{\sigma'}}] \to^*_{\mathcal{A}} \overline{C}[r\widehat{\overline{\sigma'}}] \to^*_{\mathcal{A}} \overline{C}[r\widehat{\overline{\sigma}}] = \bar{t}$. We have built a derivation: $\overline{s'} \ {}_{\mathsf{bo}(0)}\!\!\circ\!\!\!\to \overline{t'} \to^*_{\mathcal{A}} \bar{t}$. The result holds. ■

Example 4.16. Let us consider the TRS $\mathcal{S}$ built from the TRS $\mathcal{R}_1$.

$$\begin{aligned}\mathcal{S} = \{&\mathsf{f}(\#) \to \mathsf{g}(\#^1), \mathsf{f}(\#^1) \to \mathsf{g}(\#^1), \mathsf{f}(\mathsf{a}) \to \mathsf{g}(\mathsf{a}^1), \mathsf{f}(\mathsf{a}^1) \to \mathsf{g}(\mathsf{a}^1),\\ &\mathsf{f}(\mathsf{b}) \to \mathsf{g}(\mathsf{b}^1), \mathsf{f}(\mathsf{b}^1) \to \mathsf{g}(\mathsf{b}^1), \mathsf{g}(\mathsf{h}(\#)) \to \mathsf{i}(\#^1), \mathsf{g}(\mathsf{h}(\#^1)) \to \mathsf{i}(\#^1),\\ &\mathsf{g}(\mathsf{h}(\mathsf{a})) \to \mathsf{i}(\mathsf{a}^1), \mathsf{g}(\mathsf{h}(\mathsf{a}^1)) \to \mathsf{i}(\mathsf{a}^1), \mathsf{g}(\mathsf{h}(\mathsf{b})) \to \mathsf{i}(\mathsf{b}^1), \mathsf{g}(\mathsf{h}(\mathsf{b}^1)) \to \mathsf{i}(\mathsf{b}^1),\\ &\mathsf{i}(\#) \to \mathsf{a}, \mathsf{i}(\#^1) \to \mathsf{a}, \mathsf{i}(\mathsf{a}) \to \mathsf{a}, \mathsf{i}(\mathsf{a}^1) \to \mathsf{a}, \mathsf{i}(\mathsf{b}) \to \mathsf{a}, \mathsf{i}(\mathsf{b}^1) \to \mathsf{a}, \mathsf{a} \to \mathsf{b}\}.\end{aligned}$$

We have the following derivation:

$$\mathsf{g}(\mathsf{h}(\mathsf{a})) \to_{\mathcal{A},\mathsf{a}\to\#} \mathsf{g}(\mathsf{h}(\#)) \to_{\mathcal{S},\mathsf{f}(\mathsf{h}(\#))\to\mathsf{i}(\#^1))} \mathsf{i}(\#^1)).$$

In the proof of lemma 4.15, we build the derivation:

$$\mathsf{g}(\mathsf{h}(\mathsf{a})) \ {}_{\mathsf{bo}(0)}\!\!\circ\!\!\!\to_{\mathcal{R}_1,\mathsf{g}(\mathsf{h}(x))\to\mathsf{i}(x)} \mathsf{i}(\mathsf{a}^1)) \to_{\mathcal{A},\mathsf{a}^1\to\#^1} \mathsf{i}(\#^1).$$

4.4. Projecting lemma

Lemma 4.17 (projecting lemma). *Let* $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$ *be* s*-increasing, and* $\overline{s}\ {}_{\mathsf{bo}(0)}\circ\!\!\rightarrow_{\mathcal{R}} \overline{t}$. *There is a derivation:* $\mathsf{Top}(\overline{s}) \rightarrow^*_{\mathcal{A}^{\leq 1}} \rightarrow_{\mathcal{S}} \mathsf{Top}(\overline{t})$.

Proof. By definition of ${}_{\mathsf{bo}(0)}\circ\!\!\rightarrow$, there exist a context $\overline{C}[]_v$, a marked substitution $\overline{\sigma}$, and a rule
$l \rightarrow r \in \mathcal{R}$ such that $\overline{s} = \overline{C}[l\overline{\sigma}]_v$ and $\overline{t} = \overline{C}[r\widehat{\overline{\sigma}}]_v$. Since $\overline{s}$ is s-increasing, by lemma 4.6, $\overline{t}$ is s-increasing, and $\mathsf{Top}(\overline{t})$ is well defined. Moreover, the marked context $\overline{C}[]_v$, the substitution $\overline{\sigma}$, and the terms r and l are s-increasing. So, by lemmas 4.12 and 4.13: $\mathsf{Top}(\overline{s}) = \mathsf{Top}(\overline{C}[]_v)[l\mathsf{Top}(\overline{\sigma})]_v$, and, $\mathsf{Top}(\overline{t}) = \mathsf{Top}(\overline{C}[]_v)[r\mathsf{Top}(\widehat{\overline{\sigma}})]_v$. By definition of Top, $\mathsf{Top}(\overline{s}) \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\leq 1})$. Let us define the substitution $\overline{\tau}$ by $\overline{\tau} : x \mapsto \mathsf{Red}(x\mathsf{Top}(\overline{\sigma}), \mathsf{Topd}(x\mathsf{Top}(\widehat{\overline{\sigma}})))$. By definition of Red, $\mathsf{Top}(\overline{s}) \rightarrow_{\mathcal{A}} \mathsf{Top}(\overline{C}[]_v)[l\overline{\tau}]$. Moreover, $\mathsf{Top}(\overline{s}) \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\leq 1})$. Thus, we have $\mathsf{Top}(\overline{s}) \rightarrow^*_{\mathcal{A}^{\leq 1}} \mathsf{Top}(\overline{C}[]_v)[l\overline{\tau}]_v$. Let $x \in \mathcal{V}\mathsf{ar}(l)$. Let us prove that $x\overline{\tau} \in (\mathcal{F}_0 \cup \{\#\})^{\leq 1}$. Let $u \in \mathcal{P}\mathsf{os}(x\sigma)$. If $|u| \geq 1$, by definition of $\widehat{\ }$, we have $\mathsf{m}(x\widehat{\overline{\sigma}}/u) \geq 2$, and $u \notin \mathsf{Topd}(x\mathsf{Top}(\widehat{\overline{\sigma}}))$. Thus, $x\overline{\tau}$ is reduced to a constant, and since $\mathsf{Top}(\overline{s}) \in \mathcal{T}((\mathcal{F} \cup \{\#\})^{\leq 1})$, $x\overline{\tau} \in (\mathcal{F}_0 \cup \{\#\})^{\leq 1}$. Hence, the rule $l\overline{\tau} \rightarrow r\widehat{\overline{\tau}}$ belongs to $\mathcal{S}$ and $\mathsf{Top}(\overline{s}) \rightarrow^*_{\mathcal{A}^{\leq 1}} \mathsf{Top}(\overline{C})[l\overline{\tau}] \rightarrow_{\mathcal{S}} \mathsf{Top}(\overline{C})[r\widehat{\overline{\tau}}]$. By lemma 4.8, for all $x \in \mathcal{V}\mathsf{ar}(r)$,

$$x\widehat{\overline{\tau}} = \widehat{\mathsf{Red}(x\mathsf{Top}(\overline{\sigma}), \mathsf{Topd}}(x\mathsf{Top}(\widehat{\overline{\sigma}}))) = \mathsf{Red}(\widehat{x\mathsf{Top}\overline{\sigma}}), \mathsf{Topd}(x\mathsf{Top}(\widehat{\overline{\sigma}})) = x\mathsf{Top}(\widehat{\overline{\sigma}}).$$

So, $\widehat{\overline{\tau}} = \mathsf{Top}(\widehat{\overline{\sigma}})$, and $\mathsf{Top}(\overline{t}) = \mathsf{Top}(\overline{C}[]_v)[r\mathsf{Top}(\widehat{\overline{\sigma}})]_v = \mathsf{Top}(\overline{C}[]_v)[r\widehat{\overline{\tau}}]_v$. We have built a derivation: $\mathsf{Top}(\overline{s}) \rightarrow^*_{\mathcal{A}^{\leq 1}} \rightarrow_{\mathcal{S}} \mathsf{Top}(\overline{t})$. The result holds. ∎

Example 4.18. Let us consider the TRS $\mathcal{R}_1$, $\mathcal{S}$ built for this TRS, and the following $\mathsf{bo}(0)$ rewriting step: $\overline{s} = \mathsf{f}(\mathsf{f}(\mathsf{g}^1((\mathsf{a}^2)))) \circ\!\!\rightarrow_{\mathcal{R}_1, \mathsf{f}(x) \rightarrow \mathsf{g}(x)} \overline{t} = \mathsf{g}(\mathsf{f}^1(\mathsf{g}^2(\mathsf{a}^3)))$.
We have $\mathsf{Top}(\overline{s}) = \mathsf{f}(\mathsf{f}(\#^1)), \mathsf{Top}(\overline{t}) = \mathsf{g}(\#^1)$, and the following derivation:
$\mathsf{f}(\mathsf{f}(\#^1)) \rightarrow_{\mathcal{A}^{\leq 1}} \mathsf{f}(\#) \rightarrow_{\mathcal{S}, \mathsf{f}(\#) \rightarrow \mathsf{g}(\#^1)} \mathsf{g}(\#^1)$.

Definition 4.19. Let us define the relation $\preceq_{\mathsf{m}}$ on marked terms by:

$$\overline{s} \preceq_{\mathsf{m}} \overline{t} \Leftrightarrow s = t \wedge \forall u \in \mathcal{P}\mathsf{os}(s), \mathsf{m}(\overline{s}/u) < \mathsf{m}(\overline{t}/u).$$

Lemma 4.20. *Let* $\overline{s} \rightarrow^*_{\mathcal{S} \cup \mathcal{A}^{\leq 1}} \overline{t}$. *For every term* $\overline{s'} \preceq_{\mathsf{m}} \overline{s}$ *there exists a term* $\overline{t'} \preceq_{\mathsf{m}} \overline{t}$ *such that:* $\overline{s'} \rightarrow^*_{\mathcal{S} \cup \mathcal{A}^{\leq 1}} \overline{t'}$. ∎

5. Decidability of termination problems

In this section, we prove that the u-$\mathsf{bo}(k)$-termination and the $\mathsf{bo}(k)$-termination problems are decidable.

Proposition 5.1. *Let* $\overline{s_0} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$. *If the TRS* $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ *does not terminate on* $\overline{s_0}$, *then* $\mathcal{R}$ *does not* $\mathsf{bo}(0)$*-terminate on* s_0.

Proof. Assume that $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ does not terminate on $\overline{s_0} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$. By lemma 4.20, since $s_0 \preceq_{\mathsf{m}} \overline{s_0}$, there exists an infinite derivation in $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ starting from s_0. The TRS $\mathcal{A}^{\leq 1}$ is obviously u-terminating. Thus, such an infinite derivation contains an infinite number of steps in $\mathcal{S}$ and is of the form:

$$s_0 \rightarrow^*_{\mathcal{A}^{\leq 1}} \overline{s}_1 \rightarrow_{\mathcal{S}} \overline{s}_2 \rightarrow^*_{\mathcal{A}^{\leq 1}} \overline{s}_3 \rightarrow_{\mathcal{S}} \overline{s}_4 \rightarrow^*_{\mathcal{A}^{\leq 1}} \cdots \rightarrow_{\mathcal{S}} \overline{s}_{2n} \rightarrow^*_{\mathcal{A}^{\leq 1}} \cdots.$$

We now show that repeated application of lemma 4.15 yields an infinite marked $\mathsf{bo}(0)$-derivation in $\mathcal{R}$: first, consider $s_0 \to^*_{\mathcal{A}^{\leq 1}} \overline{s_1} \to_{\mathcal{S}} \overline{s_2}$. By lemma 4.15 there exists $\overline{t_1}$ such that $s_0 \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t_1} \to^*_{\mathcal{A}} \overline{s_2}$. Since $\overline{t_1} \to^*_{\mathcal{A}} \overline{s_2}$, we can apply lemma 4.15 to $\overline{t_1} \to^*_{\mathcal{A}} \overline{s_3} \to_{\mathcal{S}} \overline{s_4}$. We obtain a term $\overline{t_2}$ such that $s_0 \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t_1} \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t_2} \to^*_{\mathcal{A}} \overline{s_4}$. Following this process, we obtain an infinite sequence such that $\overline{s_0} \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t_1} \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t_2} \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \cdots \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t_n} \ldots$. We conclude that $\mathcal{R}$ does not $\mathsf{bo}(0)$-terminate on s_0. ∎

Proposition 5.2. *Let $s_0 \in \mathcal{T}(\mathcal{F})$. If $\mathcal{R}$ does not $\mathsf{bo}(0)$-terminate on s_0, then $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ does not terminate on s_0.*

Proof. If $\mathcal{R}$ does not $\mathsf{bo}(0)$-terminate on s_0, there is an infinite derivation:

$$s_0 = \overline{s}_0 \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{s}_1 \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \ldots \overline{s}_n \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \cdots .$$

The term s_0 is s-increasing since it has no mark. Moreover, the step $s_0 \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{s}_1$ is $\mathsf{bo}(0)$. By lemma 4.17, $s_0 = \mathsf{Top}(s_0) \to^*_{\mathcal{A}^{\leq 1}} \to_{\mathcal{S}} \mathsf{Top}(\overline{s_1})$. Another application of lemma 4.17 on $\overline{s_1} \;{}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{s_2}$ leads to a derivation: $\mathsf{Top}(s_0) \to^+_{\mathcal{S}\cup\mathcal{A}^{\leq 1}} \mathsf{Top}(\overline{s_1}) \to^+_{\mathcal{S}\cup\mathcal{A}^{\leq 1}} \mathsf{Top}(\overline{s_2})$. Following this process, we obtain an infinite derivation:

$$\mathsf{Top}(s_0) \to^+_{\mathcal{S}\cup\mathcal{A}^{\leq 1}} \mathsf{Top}(\overline{s_1}) \to^+_{\mathcal{S}\cup\mathcal{A}^{\leq 1}} \ldots \mathsf{Top}(\overline{s_n}) \to^+_{\mathcal{S}\cup\mathcal{A}^{\leq 1}} \cdots$$

and $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ does not terminate on $\mathsf{Top}(s_0) = s_0$. ∎

Theorem 5.3. *The $\mathsf{bo}(0)$-termination and u-$\mathsf{bo}(0)$-termination problems are decidable.*

Proof. By propositions 5.1 and 5.2, a linear TRS $\mathcal{R}$ $\mathsf{bo}(0)$-terminates on a term s_0 iff the TRS $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ terminates on s_0. If $\mathcal{R}$ does not u-$\mathsf{bo}(0)$-terminate, then by proposition 5.2, the system $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ does not u-terminate. Conversely, if $\mathcal{S} \cup \mathcal{A}^{\leq 1}$ does not u-terminate, then there exists an infinite derivation starting from a term $\overline{s_0}$. By proposition 5.1, the system $\mathcal{R}$ does not $\mathsf{bo}(0)$-terminate on s_0. So, $\mathcal{R}$ u-$\mathsf{bo}(0)$-terminates iff the ground TRS $\mathcal{S}\cup\mathcal{A}^{\leq 1}$ u-terminates. It is well known that the termination and the u-termination problems are decidable for ground TRS (see *e.g.*[1]). Hence, the $\mathsf{bo}(0)$-termination and u-$\mathsf{bo}(0)$-termination problems are decidable. ∎

Corollary 5.4. *The $\mathsf{bo}(k)$-termination and the u-$\mathsf{bo}(k)$-termination problem are decidable.*

Proof. This is a straightforward consequence of theorem 5.3 and lemma 3.12. ∎

Note that in general, for a $\mathsf{BO}(0)$ TRS, the u-$\mathsf{bo}(0)$-termination property (respectively the $\mathsf{bo}(0)$ termination property) and the u-termination (resp. termination) property are not equivalent.

Definition 5.5. Let $\mathcal{R}$ be a $\mathsf{BO}(k)$ TRS. We say that $\mathcal{R}$ has the $\mathsf{bo}(k)$ length preservation property if for every $n \in \mathbb{N}$: $\to^n_{\mathcal{R}} = \;{}_{\mathsf{bo}(k)}\!\!\to^n_{\mathcal{R}}$.

We denote by $\mathsf{BOLP}(k)$ the class of $\mathsf{BO}(k)$ TRSs that have the $\mathsf{bo}(k)$ length preservation property. Finally, the class of *bounded systems with the length preservation property* is denoted by BOLP. One can check that for every $k > 0$, $\mathsf{BOLP}(k-1) \subsetneq \mathsf{BO}(k)$.

Example 5.6. Let $\mathcal{R}_2 = \{\mathsf{f}(x) \to \mathsf{g}(x), \mathsf{g}(\mathsf{a}) \to \mathsf{f}(\mathsf{a})\}$. This TRS is $\mathsf{BO}(0)$ but does not have the $\mathsf{bo}(0)$ length preservation property. There is a derivation of length 2: $\mathsf{f}(\mathsf{a}) \to \mathsf{g}(\mathsf{a}) \to \mathsf{f}(\mathsf{a})$, but there is no $\mathsf{bo}(0)$-derivation of length 2 from $\mathsf{f}(\mathsf{a})$ to $\mathsf{f}(\mathsf{a})$ (there is one of length 0). Moreover, this TRS does not u-terminate but u-$\mathsf{bo}(0)$-terminates.

Corollary 5.7. *The termination and u-termination problems for TRSs in $\mathsf{BOLP}(k)$ are decidable.* ∎

Proof. Let $\mathcal{R} \in \mathsf{BOLP}(k)$ and let $s_0 \in \mathcal{T}(\mathcal{F})$. Clearly, if $\mathcal{R}$ does not $\mathsf{bo}(k)$-terminate on s_0, then the TRS $\mathcal{R}$ does not terminate on s_0. Conversely, let us suppose that there is an infinite derivation starting from s_0: $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} s_n \ldots$. Since $\mathcal{R}$ has the $\mathsf{bo}(k)$ length preservation property, there is for each $m \in \mathbb{N}$ a marked $\mathsf{bo}(k)$-derivation $\overline{D_m}$ such that $\overline{D_m} = s_0 \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to^m_{\mathcal{R}} \overline{s_m}$. The TRS $\mathcal{R}$ has a finite number of rules, so there is only a finite number of possible one step rewriting starting from s_0. Hence, there exists a term $\overline{s'_1}$ such that the set $\{m' \mid \overline{D_{m'}} = \overline{s_0} \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to_{\mathcal{R}} \overline{s'_1} \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to^{m'-1}_{\mathcal{R}} \overline{s_{m'}}\}$ is infinite. Repeating this process, we obtain an infinite derivation:

$$s_0 \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to_{\mathcal{R}} \overline{s'_1} \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to_{\mathcal{R}} \cdots \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to_{\mathcal{R}} \overline{s'_n} \ {}_{\mathsf{bo}(k)}\!\circ\!\!\to \ldots .$$

Hence, the TRS $\mathcal{R}$ does not $\mathsf{bo}(k)$-terminate on s_0. We have established that, for all $s_0 \in \mathcal{T}(\mathcal{F})$:

$$\mathcal{R} \ \mathsf{bo}(k)\text{-terminates on } s_0 \Leftrightarrow \mathcal{R} \text{ terminates on } s_0.$$

So, for $\mathcal{R}$, the termination problem is equivalent to the $\mathsf{bo}(k)$-termination problem, and the u-termination problem is equivalent to the u-$\mathsf{bo}(k)$-termination problem. By corollary 5.4, $\mathsf{bo}(k)$-termination and u-$\mathsf{bo}(k)$ termination problems are decidable. Hence, termination and u-termination problems for TRSs in $\mathsf{BOLP}(k)$ are decidable. ■

6. Decidability of inverse termination problems

Definition 6.1. Let $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$. We denote by $\mathsf{N}_{\overline{0}}(\overline{s})$ the number of positions $u \in \mathcal{P}\mathsf{os}(s)$ such that $\mathsf{m}(\overline{s}/u) \neq 0$: $\mathsf{N}_{\overline{0}}(\overline{s}) := \mathsf{Card}(\{u \in \mathcal{P}\mathsf{os}(s) | \mathsf{m}(\overline{s}/u) \neq 0\})$.

Lemma 6.2. *Let $\mathcal{R}$ be a linear TRS such that for all $l \to r \in \mathcal{R}$, $\mathcal{V}\mathsf{ar}(l) = \mathcal{V}\mathsf{ar}(r)$ and let $\overline{s}$,$\overline{t}$ be* s*-increasing. If $\overline{s} \ {}_{\mathsf{bo}(0)}\!\circ\!\!\to_{\mathcal{R}} \overline{t}$ then $\mathsf{N}_{\overline{0}}(\overline{s}) \leq \mathsf{N}_{\overline{0}}(\overline{t})$. Moreover, if $\mathsf{N}_{\overline{0}}(\overline{s}) = \mathsf{N}_{\overline{0}}(\overline{t})$, then $\mathsf{Top}(\overline{s}) \to_{\mathcal{S}} \mathsf{Top}(\overline{t})$ (where $\mathcal{S}$ is the ground TRS introduced in* 4.14*).* ■

Proposition 6.3. *The inverse u-*$\mathsf{bo}(k)$*-termination problem is decidable.*

Sketch of proof. By lemma 3.12, we only have to prove this result for the inverse u-$\mathsf{bo}(0)$-termination problem. Let $\mathcal{R}$ be a linear TRS. If there exists a rule $l \to r$ such that $\mathcal{V}\mathsf{ar}(r) \subset \mathcal{V}\mathsf{ar}(l)$, one can easily check that there exists an infinite inverse $\mathsf{bo}(0)$-derivation in $\mathcal{R}^{-1}$ using only the rule $r \to l$. Thus, we can suppose that $\mathcal{V}\mathsf{ar}(r) = \mathcal{V}\mathsf{ar}(l)$. Let us prove that $\mathcal{R}$ inverse u-$\mathsf{bo}(0)$-terminates iff the ground TRS S^{-1} u-terminates. Clearly, if there is an infinite derivation in S^{-1}, $\mathcal{R}^{-1}$ does not inverse $\mathsf{bo}(0)$-terminate. Conversely, let $s_0 \to_{\mathcal{R}^{-1}} s_1 \to_{\mathcal{R}^{-1}} \ldots \to_{\mathcal{R}^{-1}} s_n \to_{\mathcal{R}^{-1}} \ldots$ be an infinite inverse $\mathsf{bo}(0)$-derivation. There exists $(\overline{s_m})_{m \in \mathbb{N}}$, such that $\forall i \in \mathbb{N}$, $\overline{s_{i+1}} \ {}_{\mathsf{bo}(0)}\!\circ\!\!\to \overline{s_i}$. Without lost of generality, we can suppose that the $\overline{s_i}$ are s-increasing. By lemma 6.2, there is an integer N such that for all $m \geq N$, : $\mathsf{N}_{\overline{0}}(\overline{s_m}) = \mathsf{N}_{\overline{0}}(\overline{s_N})$. By lemma 6.2, for all $m \geq N$, $\mathsf{Top}(\overline{s_{m+1}}) \to_{\mathcal{S}} \mathsf{Top}(\overline{s_m})$. Hence, there is an infinite derivation in $\mathcal{S}^{-1}$: $\mathsf{Top}(\overline{s_N}) \to_{\mathcal{S}^{-1}} \mathsf{Top}(\overline{s_{N+1}}) \to_{\mathcal{S}^{-1}} \ldots$. Since the u-termination problem for ground TRS is decidable, the result holds. ■

Proposition 6.4. *The inverse* $\mathsf{bo}(k)$*-termination problem is decidable.* ■

Lemma 6.5. *Let $\mathcal{R}$ be a* $\mathsf{BOLP}(k)$ *TRS. The system $\mathcal{R}^{-1}$ u-terminates (respectively terminates on s) iff $\mathcal{R}$ inverse u-*$\mathsf{bo}(k)$*-terminates (resp.* $\mathsf{bo}(k)$*-terminates on s).* ■

Corollary 6.6. *Let $\mathcal{R}$ be a* $\mathsf{BOLP}(k)$ *TRS. The inverse u-termination and inverse termination problems are decidable.* ■

7. BO-systems versus BU-systems

7.1. Bottom-up derivations

We now release the hypothesis that every TRS $\mathcal{R}$ satisfies $\mathsf{LHS}(\mathcal{R}) \cap \mathcal{V} = \emptyset$. The class of BO linear TRSs is closely related to the class of bottom-up TRSs BU introduced in [4] in the following sense: every BU TRS is BO, and for every BO TRS, there is an equivalent TRS which is BU. The BU TRSs are also defined using marking tools. The marked derivation used to define BU TRS will be denoted by $\rhd\!\!\rightarrow$. Let us recall some of the definitions given in [4].

The right-action $\odot$ of the monoid $(\mathbb{N}, \max, 0)$ over the set $\mathcal{F}^{\mathbb{N}}$ consists in applying the operation max on every mark: for every $\bar{t} \in \mathcal{F}^{\mathbb{N}}, n \in \mathbb{N}$,

$$\mathcal{P}\mathsf{os}(\bar{t} \odot n) := \mathcal{P}\mathsf{os}(\bar{t}), \;\; \forall u \in \mathcal{P}\mathsf{os}(\bar{t}), \mathsf{m}((\bar{t} \odot n)/u) := \max(\mathsf{m}(\bar{t}/u), n),$$
$$(\bar{t} \odot n)^0 = \bar{t}^0$$

For every linear marked term $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \mathcal{V})$ and variable $x \in \mathcal{V}\mathsf{ar}(\bar{t})$, we define:

$$M(\bar{t}, x) := \sup\{\mathsf{m}(\bar{t}/w) \mid w < \mathsf{pos}(\bar{t}, x)\} + 1. \tag{7.1}$$

Let $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$ and $t \in \mathcal{T}$, and let us suppose that $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}})$ decomposes as

$$\overline{s} = \overline{C}[\bar{l}\overline{\sigma}]_v, \quad \text{with} \quad (l, r) \in \mathcal{R}, \tag{7.2}$$

for some marked context $\overline{C}[]_v$ and substitution $\overline{\sigma}$. We define a new marked substitution $\overline{\overline{\sigma}}$ (such that $\overline{\overline{\sigma}}^0 = \overline{\sigma}^0$) by: for every $x \in \mathcal{V}\mathsf{ar}(r)$,

$$x\overline{\overline{\sigma}} := (x\overline{\sigma}) \odot M(\bar{l}, x). \tag{7.3}$$

We then write $\overline{s} \rhd\!\!\rightarrow \bar{t}$ when

$$\overline{s} = \overline{C}[\bar{l}\overline{\sigma}], \;\; \bar{t} = \overline{C}[r\overline{\overline{\sigma}}]. \tag{7.4}$$

The map $\overline{s} \mapsto \overline{s}^0$ (from marked terms to unmarked terms) extends into a map from marked derivations to unmarked derivations: every derivation $\overline{d}$:

$$\overline{s}_0 = \overline{C}_0[\bar{l}_0\overline{\sigma}_0]_{v_0} \rhd\!\!\rightarrow \overline{C}_0[r_0\overline{\overline{\sigma}}_0]_{v_0} = \overline{s}_1 \rhd\!\!\rightarrow \ldots \rhd\!\!\rightarrow \overline{C}_{n-1}[r_{n-1}\overline{\overline{\sigma}}_{n-1}]_{v_{n-1}} = \overline{s}_n \tag{7.5}$$

is mapped to the derivation d:

$$s_0 = C_0[l_0\sigma_0]_{v_0} \to C_0[r_0\sigma_0]_{v_0} = s_1 \to \ldots \to C_{n-1}[r_{n-1}\sigma_{n-1}]_{v_{n-1}} = s_n. \tag{7.6}$$

Definition 7.1 ([4])**.** The marked derivation (7.5) is *weakly bottom-up* if, for every $0 \leq i < n$, $l_i \notin \mathcal{V} \Rightarrow \mathsf{m}(\overline{l_i}) = 0$, and $l_i \in \mathcal{V} \Rightarrow \sup\{\mathsf{m}(\overline{s_i}/u) \mid u < v_i\} = 0$.

Definition 7.2 ([4])**.** The derivation (7.6) is *weakly bottom-up* if the corresponding marked derivation (7.5) starting from the same term $\overline{s} = s$ is *weakly bottom-up*.

We shall abbreviate "weakly bottom-up" to wbu.

Definition 7.3 ([4])**.** A derivation is $\mathsf{bu}(k)$ if it is wbu and, in the corresponding marked derivation $\forall 0 \leq i \leq n$, $\mathsf{mmax}(\overline{s_i}) \leq k$.

7.2. Bottom-up systems

We denote by $\mathsf{BU}(k)$ the class of $\mathsf{bu}(k)$ systems. We define the class of *bottom-up systems*, denoted BU, by: $\mathsf{BU} = \bigcup_{k\in\mathbb{N}} \mathsf{BU}(k)$. A system is said to be strongly $\mathsf{bu}(k)$ iff every wbu derivation is $\mathsf{bu}(k)$. The class of strongly $\mathsf{BU}(k)$ systems is denoted by $\mathsf{SBU}(k)$. We define *strongly bottom-up systems*, denoted SBU by: $\mathsf{SBU} = \bigcup_{k\in\mathbb{N}} \mathsf{SBU}(k)$.

7.3. Equivalence between bounded rewriting and bottom-up rewriting

Proposition 7.4. *Let $\mathcal{R}$ be a TRS, let $\mathsf{e} = \max(\{dpt(l) | l \to r \in \mathcal{R}\})$ and let $k \in \mathbb{N}$. The following assertions hold:*

(1) *if $\mathcal{R}$ is $\mathsf{BU}(k)$, then $\mathcal{R}$ is $\mathsf{BO}(k\cdot\mathsf{e})$,*
(2) *if $\mathcal{R}$ is $\mathsf{SBU}(k)$ then $\mathcal{R}$ is $\mathsf{BOLP}(k\cdot\mathsf{e})$,*
(3) *if $\mathcal{R}$ is $\mathsf{BO}(k)$, there is an equivalent TRS $\mathcal{R}'$ in $\mathsf{BU}(1)$.* ∎

Definition 7.5. A TRS $(\mathcal{R}, \mathcal{F})$ is said to *inverse-preserves rationality* if for every recognizable set $T \subseteq \mathcal{T}(\mathcal{F})$, the set $(\to^*_{\mathcal{R}})[T] := \{s \in \mathcal{T}(\mathcal{F}) \mid \exists t \in T, s \to^*_{\mathcal{R}} t\}$ is recognizable too.

From the equivalence between BU and BO and the inverse-preservation of rationality by BU TRSs [4] we obtain:

Proposition 7.6. *Every BO TRS inverse-preserves rationality.* ∎

7.4. Classes of systems in BOLP

The class $\mathsf{SBU}(1)$ contains several classes of TRSs [4]. Among them, there are:

- the inverse left-basic semi-Thue systems (viewed as unary term rewriting systems) [12],
- the linear growing term rewriting systems [8],
- the inverse Linear-Finite-Path-Overlapping TRSs [13],
- the strongly bottom-up TRSs [4].

By corollaries 5.7 and 6.6, for all these TRSs, the termination, u-termination, inverse termination, and u-inverse termination problems are decidable.

8. Related works and perspectives

Related works. We borrowed from [4] the idea of simulating derivations according to a special strategy by some ground TRS. Note however, that the class $\mathsf{BO}(k)$ itself is new. Its advantages over the class $\mathsf{BU}(k)$ is that its definition is simpler, it allows a simpler proof of the projecting lemma and it makes lemma 3.12 true, while this lemma, mutatis mutandis, does not hold for the class $\mathsf{BU}(k)$. The principle of replacing the original rewriting relation over a signature $\mathcal{F}$ by some other binary relation over a marked-alphabet $\mathcal{F}^{\mathbb{N}}$ was already used in [5] in order to get an algorithm for termination. However, the two marking mechanisms turn out to be different:
- in the case of *word rewriting* systems, the marked derivation used here is not generated by a semi-Thue system while the marked derivation of [5] is generated by an (infinite) semi-Thue system;

- the direct image of a rational set R by a system which is match-bounded over R is rational while the direct image of a rational set by a $\mathsf{BO}(0)$ system needs not be rational; from this point of view our $\mathsf{BO}(0)$-semi-Thue systems resemble the inverses of match-bounded systems (though, they are not comparable for inclusion);
- the marking process used here extends naturally to terms while the notion of [5] seems more difficult to extend to terms (although interesting ways of doing such an extension have been studied in [6] and successfully implemented).

Perspectives. Let us mention some natural perspectives of development for this work:

- it is tempting to extend the notion of *bounded* rewriting (resp. system) to left-linear systems; this class would extend the class of growing systems studied in [11];
- we think that the direct image of a context-free language through *bounded* rewriting is context-free;
- one should try to devise a class of semi-Thue systems that includes both the class of $\mathsf{BO}(k)$ systems and the class of inverses of match-bounded systems, and still possesses the interesting algorithmic properties of these classes.

Some work in these directions has been undertaken by the authors.

Acknowledgment. We thank the anonymous referees for their useful comments, which improved the presentation of our results.

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[2] M. Dauchet. Simulation of Turing machines by a regular rewrite rule. *Theoret. Comput. Sci.*, 103(2):409–420, 1992.

[3] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In *Handbook of theoretical computer science, vol.B, Chapter 2*, pages 243–320. Elsevier, 1991.

[4] I. Durand and G. Sénizergues. Bottom-up rewriting is inverse recognizability preserving. In *Proceedings RTA'07*, volume 4533 of *LNCS*, pages 114–132. Springer-Verlag, 2007.

[5] A. Geser, D. Hofbauer, and J. Waldmann. Termination proofs for string rewriting systems via inverse match-bounds. *J. Automat. Reason.*, 34(4):365–385, 2005.

[6] A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Inform. and Comput.*, 205(4):512–534, 2007.

[7] G. Huet and D. Lankford. *On the uniform halting problem for term rewriting systems.* Rapport Laboria, 1978.

[8] F. Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, volume 1103, pages 362–376, 1996.

[9] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Vol. 2*, pages 1–116. Oxford University Press, 1992.

[10] Y. Matiyasevich and G. Sénizergues. Decision problems for semi-Thue systems with a few rules. *Theoret. Comput. Sci.*, 330(1):145–169, 2005.

[11] T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In *RTA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 256–270, London, UK, 1999. Springer-Verlag.

[12] J. Sakarovitch. *Syntaxe des langages de Chomsky, essai sur le determinisme.* Thèse de doctorat d'État, Université Paris VII, 1979.

[13] T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *RTA*, pages 246–260, 2000.

[14] Terese. *Term Rewriting Systems.* Cambridge University Press, 2003.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 357-372
http://rewriting.loria.fr/rta/

POLYNOMIALLY BOUNDED MATRIX INTERPRETATIONS

JOHANNES WALDMANN

Hochschule für Technik, Wirtschaft und Kultur (FH) Leipzig, Fakultät IMN, PF 30 11 66, D-04251 Leipzig, Germany.
E-mail address: waldmann@imn.htwk-leipzig.de

Abstract. Matrix interpretations can be used to bound the derivational complexity of rewrite systems. We present a criterion that completely characterizes matrix interpretations that are polynomially bounded. It includes the method of upper triangular interpretations as a special case, and we prove that the inclusion is strict. The criterion can be expressed as a finite domain constraint system. It translates to a Boolean constraint system with a size that is polynomial in the dimension of the interpretation. We report on performance of an implementation.

1. Introduction

Algorithms with polynomial complexity are widely accepted as practical. Since rewriting is a model of computation, we are interested in polynomial derivational complexity of rewriting. The derivational complexity of a (terminating) rewrite system is the length of a longest derivation in the system, measured as a function of the size of its initial term.

For a given terminating rewrite system, one can estimate its derivational complexity by looking at the proof method that established termination. We investigate the method of matrix interpretations [Hof06, End08]. If a rewrite system admits a matrix interpretation that is strictly compatible with all rules, then its derivational complexity is at most exponential. By restricting the shape of the matrices, we can lower this bound: if the matrices are upper triangular, derivational complexity is polynomial [Mos08].

Matrix interpretations are in fact weighted finite tree automata [Wal09], and an upper bound on the derivational complexity of the rewrite system is obtained from a bound for the growth of of the weight function computed by the automaton. So it is natural to use automata-theoretic results for a more detailed analysis. We can apply methods for the determination of (non-)ambiguity of classical (non-weighted) automata. The connection is immediate since a weighted automaton (over the standard semi-ring of natural numbers with standard addition and multiplication) can be seen as a path-counting device for an underlying classical automaton.

1998 ACM Subject Classification: F.4.2 [Grammars and Other Rewriting Systems] Term Rewriting, F.1.1 [Models of Computation] Weighted Automata, F.1.3 [Complexity Measures and Classes] Machine-independent Complexity, D.3.3 [Language Constructs and Features] Constraints.

Key words and phrases: derivational complexity of rewriting, matrix interpretation, weighted automata, ambiguity of automata, finite domain constraints.

© Johannes Waldmann
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.357

The core of the paper is organized as follows. We show in Section 3 that it is enough to consider weighted word automata even if the object is term rewriting. In Section 4 we reduce the question of growth of a weighted automaton to the question of ambiguity of non-weighted automata. Then in Section 5 we give an algorithm that decides polynomial growth of an automaton. In Section 6 we show a rewriting system that has a polynomially bounded matrix interpretation, but no triangular matrix interpretation. In Section 7 we discuss the degree of the polynomial growth bounds. We then explain in Section 8 how the decision algorithms can be realized by finite domain constraint systems, and how these can be transformed to constraints in propositional logic. In Section 9 we report on the performance of an implementation of our method.

2. Notation and Preliminaries

We recall some notions and notations.

Terms and Rewriting [Baa98]. For a ranked signature $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_k$, we denote by $\mathsf{Term}(\Sigma, V)$ the set of terms over Σ with variables from a set V, and $\mathsf{Term}(\Sigma) := \mathsf{Term}(\Sigma, \emptyset)$. The size of a ground term is the total number of symbol occurrences: if $f \in \Sigma_k$, then $|f(t_1, \ldots, t_k)| = 1 + |t_1| + \ldots + |t_k|$.

We use paths to address subterms. A path is a sequence of steps, and a step is a pair of a function symbol and a number. The number indicates in which subtree the path continues. Formally, from the given ranked signature Σ, we construct a path signature $\Sigma' \subseteq \Sigma \times \mathbb{N}$ of unary symbols: for each $f \in \Sigma$ of arity k, we have symbols $(f, 1), \ldots, (f, k)$ in Σ'. We often abbreviate (f, i) by f_i. For $t \in \mathsf{Term}(\Sigma, V)$, the set of all paths from the root of t to any node is

$$\mathsf{Path}(f(t_1, \ldots, t_k)) = \{\epsilon\} \cup \bigcup \{(f, i) \cdot p \mid 1 \le i \le k, p \in \mathsf{Path}(t_i)\}.$$

E.g., $\mathsf{Path}(f(a, g(b))) = \{\epsilon, (f, 1), (f, 2), (f, 2)(g, 1)\}$. Note that $|t| = |\mathsf{Path}(t)|$, the size of t is the number of paths in t. We write t_p for the function symbol that is reached by following the path $p \in \mathsf{Path}(t)$:

$$(f(\ldots))_\epsilon = f, \; (f(t_1, \ldots, t_k))_{(f,i)\cdot w} = (t_i)_w.$$

A rewrite system R is a set of pairs of terms with variables, and it defines a rewrite relation $\to_R$ in the usual way.

The *derivational complexity* [Hof89] of a terminating rewrite system R is the function

$$\mathsf{dc}_R : \mathbb{N}_+ \to \mathbb{N} : n \mapsto \max\{k \mid \exists t_1, t_2 \in \mathsf{Term}(\Sigma) : |t_1| \le n \wedge t_1 \to_R^k t_2\}.$$

Matrix Interpretations [End08]. Let $\mathbb{N}^d$ denote the set of d-dimensional vectors with entries in $\mathbb{N}$. We picture these as column vectors. We use these orders on $\mathbb{N}^d$:

$$x \ge y \iff x_1 \ge y_1 \wedge \ldots \wedge x_d \ge y_d, \quad x > y \iff x \ge y \wedge x_1 > y_1.$$

We use k-ary linear functions $F : (\mathbb{N}^d)^k \to \mathbb{N}^d$ that are given by k square matrices $M_1, \ldots, M_k$ and a vector v via

$$F : (x_1, \ldots, x_k) \mapsto M_1 x_1 + \ldots + M_k x_k + v.$$

We call v the *absolute part* of F, and write $v = \mathsf{abs}(F)$. A linear function is *monotone* (with respect to $>$, in each argument separately) iff for each i, the top left entry of M_i is ≥ 1.

We define orderings on these functions. For F given by $(M_1, \ldots, M_k, v)$ and F' given by $(M'_1, \ldots, M'_k, v')$, we write

$$F \geq F' \iff v \geq v' \wedge \forall 1 \leq i \leq k : M_i \geq M'_i$$
$$F > F' \iff v > v' \wedge F \geq F'$$

For any tuple of argument vectors $\vec{x} = (x_1, \ldots, x_k)$, we have $F \geq F' \Rightarrow F(\vec{x}) \geq F'(\vec{x})$ and $F > F' \Rightarrow F(\vec{x}) > F'(\vec{x})$.

A matrix interpretation assigns to each k-ary function symbol $f \in \Sigma_k$ a k-ary linear function $[f] : (\mathbb{N}^d)^k \to \mathbb{N}^d$. Since linear functions of this shape are closed with respect to composition (substitution), an interpretation can be extended from function symbols to terms (with variables).

We say an interpretation $[\cdot]$ is *compatible* with a rewrite rule $l \to r$ iff $[l] > [r]$.

Example 2.1. Take $\Sigma = \Sigma_1 = \{a, b\}$, and the monotone interpretation

$$[a] : x \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} x, \quad [b] : x \mapsto \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

that is compatible with $R = \{ab \to ba\}$, since

$$[ab] : x \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad > \quad [ba] : x \mapsto \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \qquad \blacksquare$$

If a monotone interpretation is compatible with each rule of a rewrite system R, then $t_1 \to t_2$ implies $[t_1] > [t_2]$ and since $>$ is well-founded on $\mathbb{N}^d$, the system R is terminating. More specifically, the length of each rewrite sequence starting in some $t \in \mathsf{Term}(\Sigma)$ is bounded by the first (top) component of $[t]$. This follows from the definition of $>$ on $\mathbb{N}^d$. We define the growth of the matrix interpretation $[\cdot]$ by $\mathsf{growth}_{[]} : n \mapsto \max\{[t]_1 \mid t \in \mathsf{Term}(\Sigma), |t| \leq n\}$. Then the derivational complexity of a rewriting system R is bounded by the growth of any matrix interpretation that is compatible with R.

Weighted Automata [Dro09]. We use automata with weights in $\mathbb{N}$, corresponding to matrix interpretations. We only need word (not tree) automata.

A $\mathbb{N}$-weighted word automaton $A = (Q, \lambda, \mu, \delta)$ over signature Σ consists of mappings

$$\lambda : Q \to \mathbb{N}, \mu : \Sigma \to Q^2 \to \mathbb{N}, \delta : Q \to \mathbb{N},$$

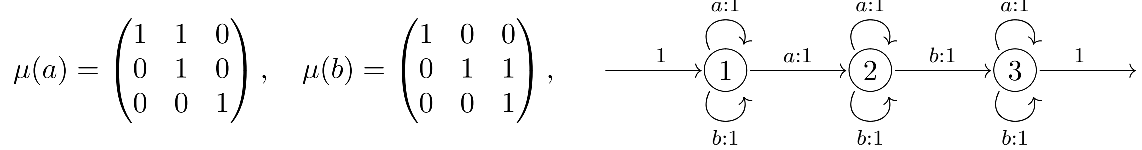
where we picture states as numbers, $Q = \{1, \ldots, d\}$, and $\lambda \in \mathbb{N}^{1 \times d}$ is the (row) vector of *initial* weights, for each letter $c \in \Sigma$, $\mu(c) \in \mathbb{N}^{d \times d}$ is a (square) *transition* matrix, and $\delta \in \mathbb{N}^{d \times 1}$ is the (column) vector of *final* weights. We extend μ homomorphically from letters to words by $\mu(u \cdot v) = \mu(u) \cdot \mu(v)$. For a word $w \in \Sigma^*$, we denote by $A(p, w, q)$ the entry at position (p, q) in the matrix $\mu(w)$. If $a = A(p, w, q)$, then we also write $p \xrightarrow{w:a}_A q$, and we define $p \xrightarrow{w}_A q$ as $A(p, w, q) > 0$. The weight $A(w)$ computed by A for a word $w \in \Sigma^*$ is given by $\lambda \cdot \mu(w) \cdot \delta$. The *growth function* growth_A of an $\mathbb{N}$-weighted automaton A over Σ is defined as the function $n \mapsto \max\{A(w) \mid w \in \Sigma^n\}$.

For a signature of unary function symbols (as in string rewriting), a d-dimensional matrix interpretation is a weighted automaton in this sense. It has states $Q = \{1, \ldots, d, d+1\}$. We have $\lambda = (1, 0, \ldots, 0)$ (the initial state is 1) and $\delta = (0, \ldots, 0, 1)^T$ (the final state is $d + 1$), and for $c \in \Sigma$, we construct $\mu(c)$ as follows: The interpretation of c is given by $[c] : x \mapsto M_1 x + v$, for a matrix $M_1 \in \mathbb{N}^{d \times d}$ and a vector $v \in \mathbb{N}^d$. From that we define

$\mu(c) = \begin{pmatrix} M & v \\ 0\ldots0 & 1 \end{pmatrix} \in \mathbb{N}^{(d+1)\times(d+1)}$. Then for any $w \in \Sigma^*$, the weight $A(w)$ computed by the automaton is equal to the first (top) entry of the value $[w]$ of w under the interpretation.

Example 2.2. [continued] The transition matrices of the automaton are given on the left, and a pictorial representation is shown on the right:

$$\mu(a) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \mu(b) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix},$$



One can now see that A (as a classical automaton) corresponds to the regular expression $\Sigma^* a \Sigma^* b \Sigma^*$, and as a weighted automaton it computes, for input w, the number of index pairs (i, j) with $i < j$ such that $w_i = a \wedge w_j = b$, equivalently, the number of inversions (with respect to $b < a$). This is exactly the function that is needed in the termination proof of $R = \{ab \to ba\}$. ■

3. Terms and Words

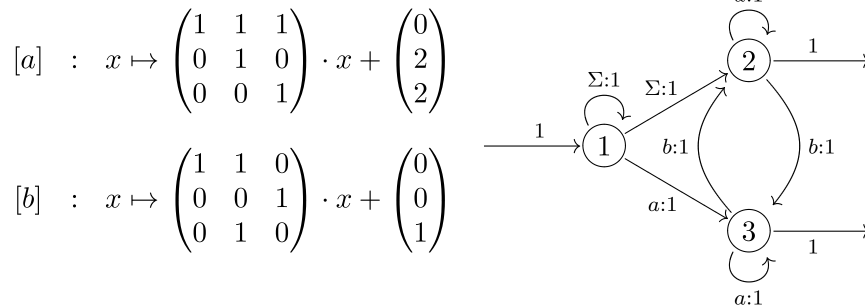
In this section we show that in order to bound the growth of a matrix interpretation (for a term rewriting system), it is sufficient to bound the growth of a $\mathbb{N}$-weighted word automaton. The reason is that a matrix interpretation corresponds to a rather restricted form of tree automaton, called *path-separated* [Kop09].

From a d-dimensional matrix interpretation $[\cdot]$ over Σ we construct a weighted word automaton A over the path signature $\Sigma' \subset \Sigma \times \mathbb{N}$ with states $Q_A = \{1, \ldots, d\}$ by taking F_i (the matrix that is the factor for the i-th argument in the linear function $[f]$) as the transition matrix $\mu_A(f_i)$. The initial weight vector λ_A is $(1, 0, \ldots, 0)$, and the final weight vector δ_A is obtained as follows. Denote by S the set of absolute parts of the interpretation $\{\mathsf{abs}[f] \mid f \in \Sigma\}$. Then $\delta_A(i)$ is 1 if there is some $v \in S$ with $v(i) > 0$. Otherwise, $\delta_A(i) = 0$. This automaton A can be used to bound the growth of the first (top) component $[t]_1$ of the interpretation of a term t.

Example 3.1. From the interpretation (for the unary signature $\{a, b\}$) on the left, we construct the automaton on the right:

$$[a] \;:\; x \mapsto \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix}$$

$$[b] \;:\; x \mapsto \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$



The final weight vector (indicated by outgoing arrows) is $\delta_A = (0, 1, 1)^T$. State 1 is not final because the top components of both absolute parts are zero. Note that the absolute parts of the interpretation are ignored except for their signum.

■

The following proposition formalizes an argument given in [Mos08] (before Theorem 6).

Proposition 3.2. *For a matrix interpretation* $[\cdot]$ *and the corresponding automaton* A*, there is a constant* C *such that for all* $t \in \mathsf{Term}(\Sigma)$ *we have* $[t]_1 \leq |t| \cdot C \cdot \mathsf{growth}_A(|t|)$.

Proof. By distributivity of matrix multiplication (over addition), the value of the matrix interpretation of a term t can be written as the sum of the values of matrix products along paths—and that is exactly what the automaton A computes:

$$[t] = \sum_{p \in \mathsf{Path}(t)} \mu_A(p) \cdot \mathsf{abs}[t_p].$$

We take C as the maximal entry of vectors in S. Then each $v \in S$ is point-wise smaller or equal to $C \cdot \delta_A$. Taking the first (top) component of $[t]$ corresponds to multiplication by λ_A from the left. In all, $[t]_1 = \lambda_A \cdot [t] \leq \sum_{p \in \mathsf{Path}(t)} C \cdot A(p) \leq \sum_{p \in \mathsf{Path}(t)} C \cdot \mathsf{growth}_A(|p|) \leq |t| \cdot C \cdot \mathsf{growth}_A(|t|)$, since $|p| \leq |t|$ (the length of a path compared to the size of the term). So the claim follows. ■

Since Proposition 3.2 introduces a factor $|t|$, we obtain the following

Theorem 3.3. *If* growth_A *(constructed from the matrix interpretation* $[\cdot]$*) is bounded by a polynomial of degree* g*, then* $\mathsf{growth}_{[]}$ *is bounded by a polynomial of degree* $g + 1$. ■

We also have a converse. For any $p \in \Sigma'^*$, there is a set T of terms $t \in \mathsf{Term}(\Sigma)$ with $|t| \leq D(1 + |p|)$ and $p \in \mathsf{Path}(t)$ and $\delta_A \leq \sum_{t \in T} \mathsf{abs}[t_p]$. Here, D is the maximal arity of Σ, and $|T| \leq d$, the dimension of the interpretation. The terms in T have the path p as their "spine", and some additional nullary symbols. At the end of the spine, there is some symbol to "cover" some non-zero entry of δ_A. Then $A(p) \leq \sum_{t \in T} \mu_A(p) \cdot \mathsf{abs}[t_p] \leq \sum_{t \in T} [t]_1$. That is, $\mathsf{growth}_A(n) \leq |T| \cdot \mathsf{growth}_{[]}(D(1 + n))$. If $\mathsf{growth}_{[]}$ is polynomially bounded, then growth_A is polynomially bounded.

Remark 3.4. The given translation ignores the entries in the absolute parts of the matrix interpretation. Indeed they do not influence the degree of the growth polynomial. Referring to Example 2.2, the present construction would remove state 3 which effectively acts as a "sink" state (no transition leaves this state). One may wonder whether state 1 could be ignored as well. In general, this may alter the degree of growth since there could be transitions from states > 1 to state 1.

4. Growth and Ambiguity

By Theorem 3.3, we will restrict our attention to weighted word automata. In the present section, we connect the weight function of a weighted automaton to the ambiguity of a non-weighted automaton. The *ambiguity* of a (non-weighted) automaton A is the function amb_A that maps each word $w \in \Sigma^*$ to the number of accepting computations (paths) of A on w.

Definition 4.1. Let A by an $\mathbb{N}$-weighted automaton. Obtain the skeleton $A' = \mathsf{skel}(A)$ by removing all weights. That is, (p, q) is an edge in $\mathsf{skel}(A)$ with label $c \in \Sigma$ exactly if $A(p, c, q) > 0$. If $\lambda_A(p) > 0$, then p is initial in A. If $\delta_A(p) > 0$, then p is final in A.

The following observation is immediate.

Proposition 4.2. *If all weights in A are from the set $\{0,1\}$, then the growth function of A and the ambiguity function of $\mathsf{skel}(A)$ are identical.*

Proof. We use the fact that $A(p,w,q)$ is the sum over the weights of all paths from p to q labelled w. Given the precondition of the proposition, the weight of a such a path in A is 1 or 0, respectively, if there exists a corresponding path in $\mathsf{skel}(A)$ or not, respectively. ∎

We observe now that the weight of an edge can be ignored if it is used at most once. To this end we define:

Definition 4.3. An edge (p,q) in $\mathsf{skel}(A)$ is called *recurrent* if there is a path from q to p in $\mathsf{skel}(A)$. All other edges are called *transitional*.

We will apply this notion to edges of weight > 0 in A as well.

Proposition 4.4. *Each path $p \stackrel{w}{\to} q$ uses each transitional edge of A at most once.*

Proof. Assume $p \to q$ is used twice. Then $p \to q \to^* p \to q$, and (p,q) is recurrent. ∎

Theorem 4.5. *The growth function of A behaves the same (up to a constant factor) as the growth function of the automaton A' obtained from A by giving weight 1 to each transitional edge.*

Proof. It is immediate that for all p,w,q: $A(p,w,q) \geq A'(p,w,q)$. Let N be the number of transitional edges in A. If $N=0$, then there is nothing to show. For $N>0$, let W be the maximal weight of a transitional edge. We claim that $A(p,w,q) \leq W^N \cdot A'(p,w,q)$. This follows since in each path, each transitional edge can be used at most once, and it contributes W (multiplicatively). ∎

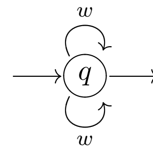
It is possible to reduce this constant W^N but we do not need this here.

We will see (Theorem 5.2, Item 1) that for A to be polynomially bounded, its recurrent edges have to have weight 1 (and not larger). Together with Proposition 4.5 this means that when we talk about polynomially bounded $\mathbb{N}$-weighted automata, we do not actually consider their edge weights in an essential way. This allows to apply known results on ambiguity, by Proposition 4.2.

5. Deciding Polynomial Growth

In this section, we give an algorithm to decide whether a given $\mathbb{N}$-weighted automaton has polynomially bounded behaviour. By applying the ideas from Section 4, we transform the problem to a question of ambiguity of non-weighted automata that can be solved with known methods. In particular we will apply

Theorem 5.1 ([Web91], Condition EDA)**.** *A trim automaton A over Σ is exponentially ambiguous if and only if there exist a state q of A and a word $w \in \Sigma^+$ such that there are at least two different paths $q \stackrel{w}{\to}_A q$.*



Here, a (classical) automaton is *trim* if each state is useful: it is accessible from some initial state, and it reaches some final state. For weighted automata A, we define the same concepts (trim, useful, accessible) by considering $\mathsf{skel}(A)$, that is, we use only paths of weight > 0.

A *strongly connected component* (SCC) of the automaton A is a maximal set C of states such that any two $p, q \in C$ are connected. Note that an edge (p, q) is recurrent (Definition 4.3) exactly if p and q belong to a common SCC.

A node is not necessarily connected to itself. Such nodes do not belong to any SCC, and they are called *transitional*. The incoming and outgoing edges of these nodes are transitional edges, as defined earlier. The automaton has a unique decomposition into SCCs and transitional nodes.

We call an automaton A *unambiguous* if it "contains no diamond" [Béa08]: there are no two paths with identical origin, end, and label.

We now characterize growth properties of $\mathbb{N}$-weighted automata:

Theorem 5.2. *For a trim* $\mathbb{N}$*-weighted automaton over* Σ,

(1) *if there is a recurrent edge with weight* > 1, *then* $\mathsf{growth}(A)$ *is exponential.*
(2) *if all recurring edges have weight one, and there is one SCC that is an ambiguous automaton, then* $\mathsf{growth}(A)$ *is exponential.*
(3) *if all recurring edges have weight one and each SCC is an unambiguous automaton, then* $\mathsf{growth}(A)$ *is bounded by a polynomial.*

Proof. Item 1: assume there is some edge $p \stackrel{x:a}{\to} q$ with $a \geq 2$ in some SCC C. Since the edge is inside an SCC, there is also a path $q \stackrel{w:b}{\to} q$ with $b > 0$. Since the automaton is trim, there is a path $i \stackrel{w_i:a_i}{\to} p$ from some initial state i, and a path $p \stackrel{w_f:a_f}{\to} f$ to some final state. Then we can compose these paths, where xw gives a loop, and we obtain that for each $k \in \mathbb{N}$, the word $w_i(xw)^k w_f$ has at least weight 2^k.

In the following cases, we apply Proposition 4.2.

Item 2: assume there is some SCC C that is ambiguous. So it contains a diamond: there are states $p, q \in C$ and a non-empty word w such that there are two different paths from p to q labelled w. Since C is strongly connected, there is a path $q \stackrel{w'}{\to} p$. This implies that the condition of Theorem 5.1 holds true (for the state p and the word $w \circ w'$), and the automaton C is exponentially ambiguous.

Item 3: follows from Remark 7.3 and Proposition 7.4 below. There we will see that in this case it does not matter that the unambiguous components are strongly connected. ∎

Example 5.3. The interpretation shown in Example 3.1 is compatible with $\{ba \to ab, a^3 \to ba^2b, b^4 \to a\}$ (SRS/Zantema/z025). The conditions of Theorem 5.2 are fulfilled: SCCs are $\{1\}$ and $\{2, 3\}$. There are no edges with weight > 1. Each SCC is unambiguous. This is trivial for the singleton, and $\{2, 3\}$ is unambiguous since the restrictions of $\mu(a)$ and $\mu(b)$ to that component are permutation matrices. Any product of permutation matrices is again a permutation matrix, and has entries in $\{0, 1\}$ only. In general, the restriction to unambiguous components does not need to give a permutation matrix. ∎

6. Comparison to Triangular Method

We prove that our method for proving polynomial derivational complexity is strictly more powerful than the method of triangular interpretations [Mos08].

We recall that the matrices in a triangular interpretation must have zeroes below the main diagonal, and zeroes or ones on the main diagonal. The elements above the main diagonal are unrestricted. The interpretation in Example 2.1 is triangular.

We make the obvious observation that each triangular interpretation fulfills the conditions of Theorem 5.2, since the SCCs of the interpretation are singletons. All edges except loops are transitional.

The interesting statement is:

Theorem 6.1. *There is a rewriting system S with these properties:*

- *S has a compatible polynomially bounded matrix interpretation,*
- *S has no compatible triangular interpretation.*

The proof is contained in the rest of this section. The main technical result is a monotonicity property of triangular interpretations (Proposition 6.2).

We use signature $\Sigma = \{L, R, a, X\}$ and take the rewriting system

$$S = \{Raa \to aaR, RX \to LX, aaL \to Laa, XL \to XRa\}.$$

This is based on a system suggested by Jörg Endrullis for a related problem.

A typical S-derivation has R travelling right, and L travelling left: for any $k \geq 0$,

$$XRa^{2k}X \to^k Xa^{2k}RX \to Xa^{2k}LX \to^k XLa^{2k}X \to XRa^{2k+1}X \to^k Xa^{2k}RaX. \quad (6.1)$$

For termination, it is essential to count the length of blocks of a modulo 2. As the above derivation shows, $XRa^{\text{even}}X \to^* XRa^{\text{odd}}X$, but $XRa^{\text{odd}}X \not\to^* XRa^{\text{even}}X$.

There is a polynomially bounded matrix interpretation that is compatible with S, see Example 7.5 below.

We now prove that S has no compatible triangular interpretation of any dimension. Since the signature is unary, we consider the transition matrices of the weighted automaton corresponding to the interpretation, cf. Example 2.1 and Example 2.2. The relations $\geq, >$ for interpretations are expressed equivalently for matrices: we write $A \geq B$ if $\forall i, j : A_{i,j} \geq B_{i,j}$, and $A > B$ if $A \geq B$ and $A_{\text{top,right}} > B_{\text{top,right}}$.

Proposition 6.2. *For any upper triangular nonnegative integer matrix A of dimension d, the sequence $(A^d, A^{d+1}, \dots)$ of powers of A is increasing: for all $k \geq d$ we have $A^k \leq A^{k+1}$.*

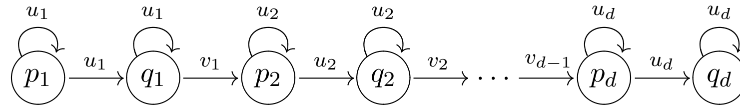
Proof. Take any $k \geq d$ and consider any pair of indices i, j with $1 \leq i, j \leq d$. Then $A^k_{i,j}$ is the sum of weights of paths of length k from i to j. For each such path p we construct a path p' (of length $k+1$) that contributes to $A^{k+1}_{i,j}$ in such a way that the construction $p \mapsto p'$ is injective and (weakly) weight-increasing. Since $|p| \geq d$, at least one vertex v is contained twice in p. Since A is upper triangular, occurrences of v must be consecutive in p. We construct p' from p by adding an edge $v \to v$ for the leftmost repeated index v of p. Since the weight of this new edge is ≥ 1, the weight of p' is at least the weight of p. By construction, the leftmost repeated vertex in p' occurs at least thrice. For any two non-equal paths p', q' constructed this way, we can delete the leftmost repetition and obtain two non-equal paths p, q of length k that contribute to $A^k_{i,j}$. ∎

Assume there is a triangular interpretation $[\cdot]$ of dimension d compatible with S. Take $k \geq d/2$. Then the interpretation must verify $[XRa^{2k}X] > [XRa^{2k+1}X]$ by the derivation 6.1. On the other hand Proposition 6.2 implies $[a^{2k}] \leq [a^{2k+1}]$. By monotonicity of multiplication, we obtain $[XRa^{2k}X] \leq [XRa^{2k+1}X]$, a contradiction. This proves Theorem 6.1.

7. Bounds for the Degree of the Growth Polynomial

The degree of ambiguity of a (classical) automaton can be determined by the following:

Theorem 7.1 ([Web91], Condition IDA$_d$). *A trim automaton A over Σ is polynomially ambiguous of degree at least d if and only if there exist states $p_1, q_1, \dots, p_d, q_d$ in A and words $u_1, \dots, u_d \in \Sigma^+$ and $v_1, \dots, v_{d-1} \in \Sigma^*$ such that $\forall 1 \le i \le d : p_i \stackrel{u_i}{\to}_A p_i \stackrel{u_i}{\to}_A q_i \stackrel{u_i}{\to}_A q_i$ and $\forall 1 \le i < d : q_i \stackrel{v_i}{\to}_A p_{i+1}$.*



It is possible to check this condition in $O(|A|^6)$ steps, and also by a corresponding constraint system of this size. Still we found it to be infeasible submit this system to a constraint solver.

The following definition fits with the constraint system that we will describe later:

Definition 7.2. An unambiguous decomposition of an automaton A with state set Q is a system $U = \{U_1, \dots, U_k\}$ of non-empty and pairwise disjoint subsets $U_i \subseteq Q$ such that

- each recurrent edge is contained in some U_i
- and for each i, the restriction of A to U_i is unambiguous.

The decomposition U defines a relation L_U on Q that consists of all pairs (p, q) such that p, q are in different components of U and there is a path from p to q.

The height of a decomposition is the height (length of a longest chain) of L_U.

Remark 7.3. The SCC decomposition in Theorem 5.2, Item 3 is an unambiguous decomposition. Its height is strictly smaller than the number of SCCs.

The connection to the degree of a polynomial growth bound is:

Proposition 7.4. *If all recurring edges of a trim $\mathbb{N}$-weighted automaton A have weight one, and* $\mathsf{skel}(A)$ *admits an unambiguous decomposition of height g, then A is polynomially bounded with degree g.*

Proof. Assume that condition IDA$_d$ holds. Then denote by P_i (Q_i, resp.) the component of the decomposition that contains p_i (q_i, resp.) Note that these components exist since both p_i and q_i are incident to recurring edges. We observe $P_i \neq Q_i$. (Otherwise, the common component would be ambiguous.) This implies $L_U(P_i, Q_i)$. We also have $P_i \neq P_{i+1}$. (Otherwise, $L_U(P_i, Q_i) \wedge L_U(Q_i, P_i)$, in contradiction to the finite height of L_U.) This implies $L_U(P_i, P_{i+1})$, and we infer $d < g$. This shows that the ambiguity of $\mathsf{skel}(A)$ is bounded by a polynomial of degree g. By Theorem 4.5, the growth function of the weighted automaton A is bounded by a polynomial of the same degree g. ∎

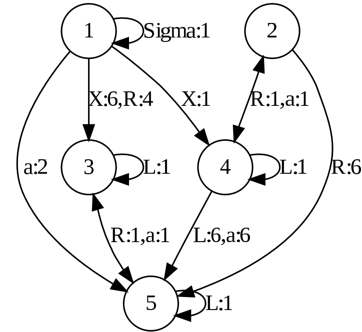
We remark that the statement in Proposition 7.4 is not an equivalence, in the sense that there may be decompositions U such that the height of L_U is larger than the degree of ambiguity. E.g., the SCC decomposition for Example 7.5.

Example 7.5. This interpretation is compatible with system S from Theorem 6.1:

$$[L] : x \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad [X] : x \mapsto \begin{pmatrix} 1 & 0 & 6 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 6 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

$$[R] : x \mapsto \begin{pmatrix} 1 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot x + \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \quad [a] : x \mapsto \begin{pmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 6 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

The translation from Section 3 turns this into a weighted automaton with 5 states, shown right. The SCCs of this automaton are $\{\{1\}, \{2, 4\}, \{3, 5\}\}$. Recurrent edges are the loops as well as the edges labelled R and a inside the two SCCs with two elements. The height of the SCC decomposition is two. Note that the only edge from 1 to $\{2, 4\}$ is labelled X, and this label does not occur inside $\{2, 4\}$. This implies that these two SCCs can be merged, resulting in the unambiguous decomposition $\{\{1, 2, 4\}, \{3, 5\}\}$. Its height is one, implying a quadratic bound on the derivational complexity of S. This bound is sharp since the rewrite system does admit derivation of quadratic length, e.g., $R^k(aa)^k \to^{k^2} (aa)^k R^k$. ∎



As an application of Proposition 7.4, we show how to modify triangular interpretations in order to improve (that is, reduce) the degree of their growth polynomial bound, in some cases.

Proposition 7.6. *For an upper triangular interpretation over alphabet* Σ*, let* D *be the set of indices* p *such that there exists* $c \in \Sigma$ *such that the entry at position* p *on the main diagonal of the linear term of the interpretation of* c *is positive. Then the interpretation is polynomially bounded with a degree of at most* $|D|$*.*

Proof. Each $p \in D$ constitutes a singleton SCC in the automaton. So all chains have length $\leq |D| - 1$, and by Theorem 3.3 the result follows. ∎

Example 7.7. (SRS/Zantema/z025) $\{ba \to ab, a^3 \to ba^2b, b^4 \to a\}$ is solved by the interpretation

$$[a] : x \mapsto \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix}, \quad [b] : x \mapsto \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot x + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

where $D = \{1, 4\}$, from which we infer maximum degree $|D| = 2$. This bound is sharp since there are derivations of quadratic length because of the rule $ba \to ab$. ∎

8. Certificates for Polynomial Bounds

In this part of the paper we show how the conditions of Theorem 5.2 and Proposition 7.4 can be deduced from existence of a "certificate". We describe how to construct a (finite domain) constraint system that specifies validity of the certificate. The subjects of the constraints are relations on the set Q of states of the given $\mathbb{N}$-weighted (word) automaton A. These unknown relations are existentially quantified at the outer level.

We also build a constraint system that describes compatibility of an (unknown) linear interpretation $[\cdot]$ with a given rewrite system R.

We combine both systems and use a constraint solver to find the interpretation $[\cdot]$ and its polynomial growth certificate A at the same time.

The (by now) standard idea for constraint solving is to translate the constraint system into a propositional logic formula. In the present paper, we discuss this "SAT encoding" of the relational constraints only. For the encoding of compatibility constraints, we refer to [End08].

8.1. Encoding of Relations

A relation on finite domains is directly modelled as a matrix of propositional variables.

In the constraint systems we will need the identity relation, on some domain. We denote it by 1, and model it trivially with a matrix of propositional constants (True on the diagonal).

For relations R, S, the implication $R \subseteq S$ is modelled by point-wise propositional implication between the corresponding matrix entries. Intersection $R \cap S$ is modelled by point-wise "and", and union $R \cup S$ by point-wise "or".

We also need composition of relations $R \circ S$, and this corresponds to Boolean matrix multiplication; as well as the inverse R^- of a relation, corresponding to matrix transposition.

Then, we want to describe the image and pre-image of a relation w.r.t. a set. By abuse of notation, if we have a binary relation $R \subseteq A \times B$ and a unary relation (a set) $A' \subseteq A$, we write $A' \circ R$ for $\{b \mid \exists a \in A' : (a, b) \in R\}$, similarly for $R \circ B'$ for $B' \subseteq B$. This is realized by multiplying the Boolean matrix R with the Boolean vector A' (resp., B').

The cost of most operations is proportional to the square of the matrix dimension, except for composition (multiplication), where the cost is cubic. Here "cost" refers to the formula size, or, equivalently, to the number of additional propositional variables that will be created by conversion to an equisatisfiable conjunctive normal form.

8.2. Encoding SCCs

The automaton A defines for each $a \in \Sigma$ an "edge" relation

$$\mu_{>0}(a) = \{(p, q) : \mu(a)(p, q) > 0\},$$

and a "heavy edge" relation

$$\mu_{>1}(a) = \{(p, q) : \mu(a)(p, q) > 1\}.$$

An (over-)approximation $E \subseteq Q \times Q$ of the reachability relation of the automaton A is specified by the constraints

$$\bigcup\{\mu_{>0}(a) \mid a \in \Sigma\} \subseteq E \quad \wedge \quad E \circ E \subseteq E$$

Correctness claim: For any solution E of the constraint system: If there is any path $p \xrightarrow{w}_A q$ of length $|w| > 0$ and weight > 0, then $E(p, q)$ holds.

Note that we do require transitivity, but not reflexivity.

Also, we do not model reachability exactly. This would require to specify E as the smallest relation with the given properties. This is not easily expressed in the given logic, where we only have existential quantification. Over-approximation of reachability may even be helpful, as explained in Section 7.

The relation $S \subseteq Q \times Q$ (over-)approximates "being in the same SCC": $S = E \cap E^-$.

Correctness claim: for any solution (E, S) of the constraint system,

- if $p \to q$ is a recurring edge in A, then $S(p, q)$;

In particular $\neg S(p, p)$ implies that p is a transitional node.

Condition (1) of Theorem 5.2 is modelled by the constraint

$$S \cap \bigcup\{\mu_{>1}(a) \mid a \in \Sigma\} = \emptyset.$$

Correctness claim: if the constraint system has a solution, then each recurring edge of A has weight $= 1$.

The size of the constraint system is cubic in the number of states of A, since we need composition of relations (once).

8.3. Encoding Unambiguity

As in [Web91], we use the criterion that an automaton A is unambiguous if the reachable and productive states of the cross product automaton $A \times A$ are on its diagonal [Sak03].

We define a relation $T \subseteq (Q \times Q)^2$ that (over-)approximates the edge relation of the product automaton, by

$$\{((p, p'), (q, q')) \mid \exists a \in \Sigma : \mu_{>0}(a)(p, q) \wedge \mu_{>0}(a)(p', q')\} \subseteq T.$$

We use relations $R, P \subseteq Q \times Q$ with the intention that $R(p, q)$ holds true if the state $(p, q) \in A \times A$ is reachable, and $P(p, q)$ holds true if the state $(p, q) \in A \times A$ is productive. We specify:

- the diagonal states are reachable and productive: $1 \subseteq R \wedge 1 \subseteq P$,
- each successor of a reachable state is reachable: $(R \circ T) \cap S \subseteq R$,
  note that we restrict to transitions that stay inside the (approximated) SCCs.
- each predecessor of a productive state is productive: $(T \circ P) \cap S \subseteq P$,
- each state that is both reachable and productive, is on the diagonal: $R \cap P \subseteq 1$

Correctness claim: if the constraint system has a solution, then S describes an unambiguous decomposition U of A. Here, p and q are in the same component of U if $S(p, q)$. The height of L_U is finite. — Proof: By construction, each recurrent edge is contained in some component, and each component is unambiguous. A cycle of components is impossible since S is transitive.

The size of the constraint system is $\Theta(d^4)$ since we need to compute the (pre)image of a relation on Q^2.

Remark 8.1. If we collect all the constraints up to here, then we already have a method that is more powerful in proving polynomial complexity bounds than triangular interpretations. E.g., it finds a proof for the system in Theorem 6.1. In the following, we bound the degree of the growth polynomial.

8.4. Unary Encoding of (Small) Numbers

A number n is given by a unary relation N that we view as a subset of the range. In our case, $N \subseteq \{1, \dots, d\}$ where d is the dimension of the interpretation = the number of states of the automaton. Equivalently, $N : \{1, \dots, d\} \to \mathsf{Boolean}$. A value of k is encoded by the assignment $\{1, \dots, k\} \mapsto \mathsf{True}, \{k+1, \dots, d\} \mapsto \mathsf{False}$. That is, for any N encoding a number we have the constraint $\forall 1 \leq i < d : N(i) \Leftarrow N(i+1)$.

For the given application, we did not investigate other encodings (e.g., binary) as the range of numbers is small (it is the number of states of the automaton), and we do not need arithmetical operations, only comparison: $M > N$ is given by $\bigvee\{M(i) \wedge \neg N(i) \mid 1 \leq i \leq d\}$.

8.5. Encoding the Height of a Relation

The height of a relation $L \subseteq Q \times Q$ is the length of a longest L-chain. To express the condition "the height of L is at most g", we use a "height" function $h : Q \to \{0, 1, \dots, g\}$ and the constraints

$$\forall p, q : L(p, q) \Rightarrow (h(p) > h(q)).$$

The range of h is implemented by unary numbers as discussed above.

The cost of this constraint is $\Theta(d^2 \cdot g)$, since we do d^2 comparisons that cost $\Theta(g)$ each.

We apply this for g = the intended degree of polynomial growth, and the relation

$$L = \{(p, q) \mid S(p, p) \wedge \neg S(p, q) \wedge E(p, q) \wedge S(q, q)\}.$$

Correctness claim: if the constraint system has a solution, then the automaton A admits an unambiguous decomposition of height $\leq g$. — Proof: If p, q are recurrent nodes from distinct S-components such that A contains a path from p to q, then $L(p, q)$. Therefore, each L-chain is a chain of S-components, and each of them is unambiguous.

Remark 8.2. In the general case, an unambiguous component may contain recurrent and transitional edges, and the weight of transitional edges is irrelevant by Theorem 4.5. Since we use the relation S (for efficiency of implementation), we discard the possibility that unambiguous components contain transitional edges of weight > 1.

8.6. Encoding Improved Triangular Interpretations

We show that Proposition 7.6 can be implemented as a constraint system with little effort. We use binary variables $C_1, \dots, C_d$ to encode membership in the set D:

$$\forall 1 \leq p \leq d : \forall c \in \Sigma : [c](p, p) > 0 \Rightarrow C_p$$

and we express that at most g of these variables are true, by a relation $Z \subseteq Q \times \{1, \dots, g\}$ with the intention that $Z(p, h)$ = "at most h of $C_1, \dots, C_p$ are true." This is specified by

$$Z(p, h) = (C_p \wedge Z(p-1, h-1)) \vee (\neg C_p \wedge Z(p-1, h)).$$

and obvious border cases. This constraint system is used in addition to the constraint system that describes compatibility of a triangular interpretation with the rewriting system. It is statically known that entries below the main diagonal are zero, so the multiplication of such matrices can be implemented with less effort than for full matrices. So we expect the constraint solver to be able to handle somewhat larger matrix dimensions. The interpretation in Example 7.7 was found this way.

9. Results

The method described in this paper was implemented for the termination analyzer Matchbox. Given a rewriting system R, the implementation produces constraint systems for various values of d (matrix dimension) and g (degree of growth) and submits them to solvers in parallel. As soon as a solution for some g is found, all other attempts for degrees $\geq g$ are terminated.

In the 2009 Termination Competition, Matchbox entered the category of Derivational Complexity/Full rewriting. Of the 616 problems, it could prove polynomial derivational complexity for 58 of them. The winner CaT got 98 answers. (The results of the third participant TCT are currently not available.)

In 3 cases, Matchbox got a better degree bound than CaT, and in two cases, Matchbox could prove polynomial complexity where CaT couldn't. In 9 cases, CaT got a better degree bound than Matchbox; and in 36 cases, CaT proved polynomial complexity where Matchbox couldn't.

The differences in performance are mainly due to the different methods that the programs apply: Matchbox uses the method of the present paper exclusively, while CaT uses a combination of triangular interpretations, root labelling, relative termination, arctic matrix interpretations and match-bounds.

10. Discussion

There are several open question related to polynomially bounded matrix interpretations. We mention only a few.

Formal verification of certificates for polynomial derivational complexity is possible. As a certificate, we can take the relations that are specified by the constraint system. Then it is easy to formally verify their validity, since it only needs propositional logic. (If a SAT solver can find it, then it is easy to check.)

Does there exist a rewriting system with polynomially bounded complexity that does not admit a polynomially bounded matrix interpretation? It is easy to answer "yes" for term rewriting: it was already noted in [End08] that the ground rewriting system $\{f(a,b) \to f(b,b), f(b,a) \to f(a,a)\}$ does not admit a matrix interpretation. Its derivational complexity obviously is linear. The question seems much harder for string rewriting, even in the following restricted setting:

Do all match-bounded string rewriting systems have a polynomially (even linearly) bounded matrix interpretation? Experience in the recent termination competition suggests otherwise. Still, this may be due to vastly different search methods. Certificates for match-boundedness can be found by completion [End06, Kor09], and quite often this gives large automata quickly. On the other hand, matrix interpretations are usually found via constraint solving (via SAT, as described here), and this usually cannot handle much more than dimension 5.

From a "practical" viewpoint, one would be interested in a constraint system that describes a certificate for the exact degree of the growth polynomial with less than $O(n^6)$ size—or in an altogether different method for the construction of automata that are compatible with a given rewrite system, and polynomially bounded. As very concrete challenges, one could try to find polynomially bounded matrix interpretations for two famous rewrite

systems z001= $\{a^2b^2 \to b^3a^3\}$ and z086= $\{a^2 \to bc, b^2 \to ac, c^2 \to ab\}$. In both cases, matrix interpretations (of dimension 5) are known, but they grow exponentially.

What about using different weight domains for the interpretations? It is easy to generalize the matrix interpretation method to rational (or real) numbers [Geb07], but it seems harder to obtain complexity information from such interpretations. We would need to decide whether a given $\mathbb{Q}$-weighted automaton is polynomially bounded. The easy connections from Section 4 do not hold, as a polynomially growing automaton could contain recurrent edges of weight > 1 (e.g., if they are directly followed by edges of suitable weights < 1).

Acknowledgements. I appreciate the anonymous referees' careful reading and detailed discussion.

References

[Baa98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Béa08] Marie-Pierre Béal, Eugen Czeizler, Jarkko Kari, and Dominique Perrin. Unambiguous automata. *Mathematics in Computer Science*, 1(4):625–638, 2008.

[Dro09] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata.* Springer, 2009.

[End06] Jörg Endrullis, Dieter Hofbauer, and Johannes Waldmann. Decomposing terminating rewriting relations. In *Workshop on Termination.* 2006.

[End08] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

[Geb07] Andreas Gebhardt, Dieter Hofbauer, and Johannes Waldmann. Matrix evolutions. In Dieter Hofbauer and Alexander Serebrenik (eds.), *Proc. Workshop on Termination, Paris.* 2007.

[Hof89] Dieter Hofbauer and Clemens Lautemann. Termination proofs and the length of derivations. In Nachum Dershowitz (ed.), *RTA*, *Lecture Notes in Computer Science*, vol. 355, pp. 167–177. Springer, 1989.

[Hof06] Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Frank Pfenning (ed.), *RTA*, *Lecture Notes in Computer Science*, vol. 4098, pp. 328–342. Springer, 2006.

[Kop09] Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

[Kor09] Martin Korp and Aart Middeldorp. Match-bounds revisited. *Inf. Comput.*, 207(11):1259–1283, 2009.

[Mos08] Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay (eds.), *FSTTCS*, *LIPIcs*, vol. 08004. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[Sak03] Jacques Sakarovitch. *Eléments de théorie des automates.* Vuibert, Paris, 2003.

[Wal09] Johannes Waldmann. Automatic termination. In Ralf Treinen (ed.), *RTA*, *Lecture Notes in Computer Science*, vol. 5595, pp. 1–16. Springer, 2009.

[Web91] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theor. Comput. Sci.*, 88(2):325–349, 1991.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 373-384
http://rewriting.loria.fr/rta/

OPTIMIZING MKB$_{\mathrm{TT}}$ (SYSTEM DESCRIPTION) *

SARAH WINKLER [1] AND HARUHIKO SATO [2] AND
AART MIDDELDORP [1] AND MASAHITO KURIHARA [2]

[1] Institute of Computer Science
University of Innsbruck, Austria
E-mail address, S. Winkler: sarah.winkler@uibk.ac.at

[2] Graduate School of Information Science and Technology
Hokkaido University, Japan
E-mail address, H. Sato: haru@complex.eng.hokudai.ac.jp

E-mail address, A. Middeldorp: aart.middeldorp@uibk.ac.at

E-mail address, M. Kurihara: kurihara@ist.hokudai.ac.jp

Abstract. We describe performance enhancements that have been added to mkb$_{\mathrm{TT}}$, a modern completion tool combining multi-completion with the use of termination tools.

1. Introduction

The Knuth-Bendix completion tool mkb$_{\mathrm{TT}}$ combines a multi-completion approach as introduced by Kurihara and Kondo [Kur99] with the use of automatic termination tools proposed by Wehrman, Stump and Westbrook [Weh06]. In this paper we present several performance enhancements which improved the first version described in [Sat08].

(1) Checking for and joining isomorphic processes results in considerable speedups for some input systems.
(2) Critical pair criteria were carried over to the context of multi-completion to reduce the number of nodes.
(3) Several indexing techniques were implemented to allow for a higher inference rate, namely path indexing, discrimination trees and code trees.
(4) Different selection strategies to choose the next process and node were compared.

The potential of these optimizations, which are described in the next four sections, is witnessed by the first automatic completion of the CGE$_4$ system (an axiomatization of group theory with four commuting endomorphisms) obtained with the new version of mkb$_{\mathrm{TT}}$. The optimizations can be conveniently configured through the web interface, which is described in Section 6. The experimental results reported in Section 7 show their usefulness.

Key words and phrases: Knuth-Bendix completion, termination prover, automated deduction.

*This research is supported by FWF (Austrian Science Fund) project P18763. The first author is supported by a DOC-fFORTE fellowship of the Austrian Academy of Sciences.

© S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.373

We start by recalling some basic definitions. A Knuth-Bendix completion (KB) procedure takes a set of equations together with a reduction order as input and aims to produce a terminating and confluent rewrite system with the same equational theory. We call a KB run *fair* if all critical pairs among persistent rewrite rules are eventually considered. In Knuth-Bendix completion with termination tools (KBtt), a reduction order is no longer required. Instead, the inference system of KB is extended to work on tuples (E, R, C) of a set of equations E and rewrite systems R and C, which we refer to as KBtt *states*. The inference system $\mathsf{mkb}_{\mathrm{TT}}$ simulates multiple KBtt runs. Each run is identified by a bit sequence called a *process*. The inference rules of $\mathsf{mkb}_{\mathrm{TT}}$ operate on objects called *nodes*. A node has the form $\langle s : t, R_0, R_1, E, C_0, C_1\rangle$ where s, t are terms and the remaining components (called *labels*) are sets of processes. In the sequel we assume familiarity with [Sat08, Sat09].

2. Isomorphisms on Processes

The number of parallel processes simulated by $\mathsf{mkb}_{\mathrm{TT}}$ is critical for overall performance. However, some systems exhibit process pairs which are very similar and actually equally likely to succeed. This is illustrated in the following example.

Example 2.1. When running $\mathsf{mkb}_{\mathrm{TT}}$ on CGE$_2$ [Stu06], a process p with state

$$E_p = \left\{\begin{array}{c} (x * y) * z \approx x * (y * z) \\ \mathsf{f}(\mathsf{e}) \approx \mathsf{e} \\ \mathsf{g}(\mathsf{e}) \approx \mathsf{e} \\ \mathsf{g}(x) * \mathsf{f}(y) \approx \mathsf{f}(y) * \mathsf{g}(x) \end{array}\right\} \qquad R_p = C_p = \left\{\begin{array}{c} \mathsf{e} * x \to x \\ \mathsf{g}(x) * x \to \mathsf{e} \\ \mathsf{f}(x * y) \to \mathsf{f}(x) * \mathsf{f}(y) \\ \mathsf{g}(x * y) \to \mathsf{g}(x) * \mathsf{g}(y) \end{array}\right\}$$

has to orient the equation $\mathsf{g}(x) * \mathsf{f}(y) \approx \mathsf{f}(y) * \mathsf{g}(x)$. As both orientations are possible the process will be split, but the states (E_{p0}, R_{p0}, C_{p0}) and (E_{p1}, R_{p1}, C_{p1}) of the resulting child processes are the same up to interchanging f and g. Hence the deductions corresponding to these processes will be symmetric if the choice function is reasonably fair.

The following definition formally captures such similarities in general.

Definition 2.2. A mapping $\theta\colon \mathcal{T}(\mathcal{F},\mathcal{V}) \to \mathcal{T}(\mathcal{F},\mathcal{V})$ induces an *isomorphism* between two rewrite systems R and R' if $R' = \{\theta(l) \to \theta(r) \mid l \to r \in R\}$ and for all terms s and t, $s \to_R t$ if and only if $\theta(s) \to_{R'} \theta(t)$. Similarly, θ induces an isomorphism between two sets of equations E and E' if $E' = \{\theta(u) \approx \theta(v) \mid u \approx v \in E\}$ and for all terms s and t, $s \approx_E t$ if and only if $\theta(s) \approx_{E'} \theta(t)$.

Two rewrite systems R and R' are *isomorphic* if there exists an isomorphism θ between them, which is expressed by writing $R \cong_\theta R'$. Two KBtt states (E, R, C) and (E', R', C') are isomorphic if there is an isomorphism θ such that $E \cong_\theta E'$, $R \cong_\theta R'$, and $C \cong_\theta C'$. Two $\mathsf{mkb}_{\mathrm{TT}}$ processes p and q are isomorphic in a node set N if their projected states $(E(N,p), R(N,p), C(N,p))$ and $(E(N,q), R(N,q), C(N,q))$ are isomorphic.

Isomorphic processes are equally likely to succeed, which is easy to prove as the rewrite relations coincide.

Lemma 2.3. *Assume N is a node set with isomorphic processes p and q. If there exists a fair mkb_{TT} completion run $N \vdash^* N'$ such that $E(N', p) = \varnothing$ then there is another fair deduction $N \vdash^* N''$ such that $E(N'', q) = \varnothing$.*

Due to Lemma 2.3, completeness is not compromised if one of two isomorphic processes is removed. $\mathsf{mkb}_{\mathrm{TT}}$ exploits such symmetries by checking for two concrete shapes of isomorphisms.

- *Renamings* swap function symbols as in Example 2.1 where $p0$ and $p1$ are isomorphic processes under the mapping θ that exchanges f and g.
- *Argument permutations* associate with every function symbol f of arity n a permutation $\pi_f \in \mathcal{S}_n$. Then the mapping on terms defined by $\theta(x) = x$ and $\theta(f(t_1, \ldots, t_n) = f(\theta(t_{\pi_f(1)}), \ldots, \theta(t_{\pi_f(n)}))$ may also induce an isomorphism. For example, during a completion run of SK3.02 [Ste90] a process p with state
$$E_p = \{(x + y) + z \approx x + (y + z)\} \qquad R_p = C_p = \begin{Bmatrix} \mathsf{f}(\mathsf{f}(x)) \to x \\ \mathsf{f}(x + y) \to \mathsf{f}(x) + \mathsf{f}(y) \end{Bmatrix}$$
has to orient the associativity axiom. Again both orientations are possible, but the two child processes emerging from a process split are isomorphic under an argument permutation satisfying $\pi_+ = (1\ 2)$.

$\mathsf{mkb}_{\mathrm{TT}}$ can check for both kinds of isomorphisms, either only in orient inferences to avoid unnecessary splittings or by comparing the states of all process pairs repeatedly.

3. Indexing Techniques

For rewrite inferences $\mathsf{mkb}_{\mathrm{TT}}$ needs to filter out nodes from the current node set that can be used to rewrite a given node. In the case of deduce inferences one needs to find nodes that overlap with the current node. Especially rewrite inferences are frequently executed. Performing these operations efficiently is crucial for overall performance.

In automated reasoning this problem is referred to as the *indexing problem*: Given a large set L of terms (the *index*), a binary relation on terms R (the *retrieval condition*) and a term t (the *query term*), find all $s \in L$ such that $(s, t) \in R$. For this purpose, a number of sophisticated term indexing techniques have been developed [Sek01].

In the case of $\mathsf{mkb}_{\mathrm{TT}}$, for $\mathsf{rewrite}_1$ the retrieval of variants, for $\mathsf{rewrite}_2$ finding encompassments, and for deduce the search of unifiable terms is required. Since encompassment retrieval can be implemented as multi-term retrieval of subsumed terms, indexing structures need to support variance, subsumption and unifiability as retrieval conditions. Variant and encompassment retrieval is required particularly often and consumes about 25% of the total computation time if performed naively.

$\mathsf{mkb}_{\mathrm{TT}}$ currently supports *path indexing* [McC92, Gra96] to retrieve unifiable terms. For variant and generalization retrieval, also *discrimination trees* [McC92, Gra96] and *code trees* [Vor95] are implemented.

In line with earlier observations in automated reasoning, the use of indexing techniques in $\mathsf{mkb}_{\mathrm{TT}}$ increases performance considerably.

4. Critical Pair Criteria

Maintenance and treatment of equational consequences derived in a deduction is a critical factor in many automated reasoning tools. Since it keeps track of multiple processes this is an even more serious issue for $\mathsf{mkb}_{\mathrm{TT}}$. For standard completion several *critical pair criteria* were proposed as a means to filter out equational consequences that can be ignored without losing completeness.

In the setting of $\mathsf{mkb}_{\mathrm{TT}}$, also the computation of critical pair criteria can be shared among multiple processes. If a node with datum $s : t$ is deduced from an overlap o for a process set E then $\mathrm{CPC}(o, E, N)$ returns all processes for which $s \approx t$ is not superfluous. Thus the deduce rule is modified as follows.

Definition 4.1. The inference rule

$$\textsf{deduce} \quad \frac{N}{N \cup \{\langle s : t, \varnothing, \varnothing, E, \varnothing, \varnothing\rangle\}}$$

is applicable if there exist nodes $\langle l : r, R, \dots\rangle, \langle l' : r', R', \dots\rangle \in N$ such that $s \approx t$ is a critical pair originating from an overlap o involving the rules $l_1 \to r_1$ and $l_2 \to r_2$, and $E = \mathrm{CPC}(o, R \cap R', N) \neq \varnothing$.

Given a critical pair $s \approx t$ originating from an overlap $(l_1 \to r_1, p, l_2 \to r_2)_\sigma$, all implemented criteria have in common that the term $l_1\sigma = l_1\sigma[l_2\sigma]$ is checked for being reducible in a different way than by $l_1 \to r_1$ and $l_2 \to r_2$. Consequently, all criteria need to filter the current node set N for nodes that allow to rewrite $l_1\sigma$ in an appropriate way (depending on the actual criterion). The execution of a CPC function can thus be shared among multiple processes. Moreover, CPC functions can take advantage from term indexing techniques as they require to find encompassments for a given term.

The criteria implemented in $\mathsf{mkb}_{\mathrm{TT}}$ are multi-completion variants of the criteria developed for standard completion: the primality criterion PCP [Kap88], the blocked criterion BCP [Bac88], and the connectedness criterion CCP [Küc85]. The projection of an $\mathsf{mkb}_{\mathrm{TT}}$ deduction using a critical pair criterion to a process p yields a KBtt deduction which is fair with respect to the criterion. Being specializations of the more general compositeness criterion [Bac94], the implemented criteria are *correct*. This means that a nonfailing deduction which is fair with respect to a critical pair criterion is also fair in the more general sense. Hence, according to the definition of fairness in $\mathsf{mkb}_{\mathrm{TT}}$ [Sat09, Section 4], deductions using critical pair criteria are also fair, so correctness is preserved. Experimental results evaluating the usefulness of the critical pair criteria are given in Section 7.

5. Selection Strategies

At the beginning of $\mathsf{mkb}_{\mathrm{TT}}$'s main control loop, a *choice* function selects a node to process next by evaluating a cost heuristic. The measure applied in this selection has significant impact on performance. To allow for a variety of strategies, we defined a language that is general enough to cover selection strategies that proved to be useful, but also allows to capture some concepts used in selection strategies of other automated reasoning tools. A selection strategy is thus specified by the following grammar:

$$\begin{aligned}
\textit{strategy} &::= \texttt{?} \mid \texttt{(}\textit{node_p}\texttt{,}\textit{strategy}\texttt{)} \mid \textit{float}\texttt{(}\textit{strategy}\texttt{:}\textit{strategy}\texttt{)}\\
\textit{node_p} &::= \texttt{data(}\textit{termpair_p}\texttt{)} \mid \texttt{el(}\textit{processset_p}\texttt{)} \mid \texttt{-}\,\textit{node_p} \mid \textit{node_p}\ \texttt{+}\ \textit{node_p} \mid \texttt{*}\\
\textit{processset_p} &::= \texttt{min(}\textit{process_p}\texttt{)} \mid \texttt{sum(}\textit{process_p}\texttt{)} \mid \texttt{\#}\\
\textit{process_p} &::= \textit{process_p}\ \texttt{+}\ \textit{process_p} \mid \texttt{e(}\textit{eqs_p}\texttt{)} \mid \texttt{r(}\textit{trs_p}\texttt{)} \mid \texttt{c(}\textit{trs_p}\texttt{)}\\
\textit{eqs_p} &::= \texttt{sum(}\textit{termpair_p}\texttt{)} \mid \texttt{\#}\\
\textit{trs_p} &::= \texttt{sum(}\textit{termpair_p}\texttt{)} \mid \texttt{cp(}\textit{eqs_p}\texttt{)} \mid \texttt{\#}\\
\textit{termpair_p} &::= \texttt{smax} \mid \texttt{ssum}
\end{aligned}$$

The following paragraphs shortly comment on the elements of selection strategies.

- The simplest strategy `?` chooses a node randomly. Otherwise, a strategy can be a tuple `(`np`,`s`)` consisting of a node property np and another strategy s. By using this rule multiple times, a node property tuple of the form `(`np_1`, ... (`np_k`,?) ...)` is obtained. A selection strategy is implemented by mapping each node to the corresponding cost tuple of integers and choosing the (lexicographic) minimum. To mix strategies, a strategy can also be of the shape r`(`s_1`:`s_2`)`. Here r is assumed to be a rational value in $[0, 1]$, with the intention that in every selection step the strategy s_1 is applied with probability r, and s_2 is used in the remaining cases.
- Node properties capture features of nodes with integers. A node property of a node $n = \langle s : t, \ldots, E, \ldots\rangle$ can be its creation time (denoted by `*`), a property of the node's datum $s : t$, or a process set property pp of its equation labels E, which is written as `el(`pp`)`. Moreover, node properties can be added or inverted.
- A process set property may be the number of processes it contains (denoted by `#`) or the sum or the minimum over a property of the processes it contains.
- A process property pp of a process p may be an equation set property of its equation projection $E(N, p)$ or a TRS property of either its rule projection $R(N, p)$ or its constraint projection $C(N, p)$, which is expressed by writing `e(`pp`)`, `r(`pp`)` and `c(`pp`)`, respectively. In addition, process properties can be added.
- An equation set property of a set of equations E can be its cardinality (`#`), or the sum over a term pair property of all its elements. A TRS property of a rewrite system R can additionally be a property of its critical pairs $\mathrm{CP}(R)$.
- Finally, a term pair property of terms s and t can be the sum $|s| + |t|$ or maximum $\max\{|s|, |t|\}$ of their sizes.

Many automated reasoning tools [Vor01] employ a size-age ratio when selecting a fact to be processed next. For example, if this ratio is 2 : 3 then out of 5 selections 2 will pick the oldest and 3 the smallest node, i.e., the node where the sum of its term sizes $|s| + |t|$ is minimal. In $\mathsf{mkb}_{\mathrm{TT}}$ a parameter $r \in [0, 1]$ controls the ratio of weight-determined selections, i.e., an age-weight ratio of 2 : 3 would correspond to $r = 0.6$. This is described with the following strategy:

$$s_{\texttt{size/age}}(r) = r\texttt{((data(ssum),?):(*,?))}$$

In the first version of $\mathsf{mkb}_{\mathrm{TT}}$ [Sat08] we first chose a process p for which $|E(N, p)|+|R(N, p)|$ was minimal and then a node for this process by considering the term size and timestamp, the latter to ensure fairness of the derivation. By again using a size-age ratio in the second step, this is captured by the strategy

$$s_{\texttt{mkbtt1}}(r) = \texttt{(el(min(e(\#)+r(\#))),}s_{\texttt{size/age}}(r)\texttt{)}$$

The selection approach used in Slothrop [Weh06] corresponds to choosing a process for which $|E(N, p)| + |\mathrm{CP}(R(N, p))| + |C(N, p)|$ is minimal, which is expressed as follows:

$$s_{\texttt{slothrop}}(r) = \texttt{(el(min(e(\#)+r(cp(\#))+c(\#))),}s_{\texttt{size/age}}(r)\texttt{)}$$

We recently experimented with an approach which first restricts attention to those processes where the number of symbols in $E(N, p)$ and $C(N, p)$ is minimal, then selects nodes with minimal data and finally goes for a node which has the greatest number of processes in its equation label:

$$s_{\texttt{sum}} = \texttt{(el(min(e(sum(ssum))+c(sum(ssum)))),(data(ssum),(-el(\#),?))))}$$

The strategy where `ssum` is replaced by `smax` is referred to as $s_{\mathtt{max}}$. To use other heuristics than those just described, the strategy can also be specified via a command line option.

6. Web Interface

In addition to a command line interface, mkb$_{\mathrm{TT}}$ is now also available via a web interface which allows to configure various options.

- The user can set both a global timeout and a timeout for each termination check.
- Concerning termination checks, users can either apply T$_\mathrm{T}$T$_2$ [Kor09] or incorporate an external tool. If T$_\mathrm{T}$T$_2$ is used internally, different predefined termination strategies can be selected. This includes basic reduction orderings as well as `ttt2micro` and `ttt2fast`, two powerful and fast strategies. Alternatively, a user-defined strategy may be supplied in the strategy language of T$_\mathrm{T}$T$_2$. Another option allows to check termination externally with AProVE [Gie06] or MuTerm [Ala07].
- For the retrieval of encompassments and variants, one of the implemented indexing techniques can be selected (path indexing, discrimination trees, code trees or naive search in the node set).
- To filter deduced equations, one of the three implemented critical pair criteria PCP, BCP or CCP can be selected, as well as certain combinations.
- Users can choose if isomorphism checks employing symbol renamings or term permutations are to be used, and whether these checks are performed repeatedly or only when processes are split.

The screenshot in Figure 1 gives an impression of the web interface, which is accessible at

`http://colo6-c703.uibk.ac.at/mkbtt/interface/`

7. Experimental Results

All of the following experiments were performed on a single core of a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6GHz and 64GB of main memory. As a test set we used 101 problems collected from various papers. These include theories underlying unit equality problems in TPTP 3.6.0 [Sut09], all examples coming with the Slothrop [Weh06] distribution, all systems in [Ste90, Section 3], instances of parametric systems given in [Bün94, Chr89], and systems describing commuting group endomorphisms [Stu06]. The whole testbench as well as the full experimental data can be obtained from the website.

In its fastest configuration it took the previous version [Sat08] of mkb$_{\mathrm{TT}}$ 175 seconds to complete CGE$_2$, and more than 3000 seconds to complete CGE$_3$. The implemented optimizations, in particular isomorphisms and different selection strategies, allowed for significant speedups: Using $s_{\mathtt{max}}$ as selection strategy, `ttt2micro` for termination checks and code trees for indexing operations, completing CGE$_2$ and CGE$_3$ requires 8.4 and 184 seconds, respectively. Using periodical checks for renaming isomorphisms further reduces these numbers to 4.7 and 33 seconds. The implemented optimizations even allow mkb$_{\mathrm{TT}}$ to complete CGE$_4$, which was neither achieved with the previous version nor with any other approach we know of. With the same settings as described above, a complete system is obtained in 622 seconds.

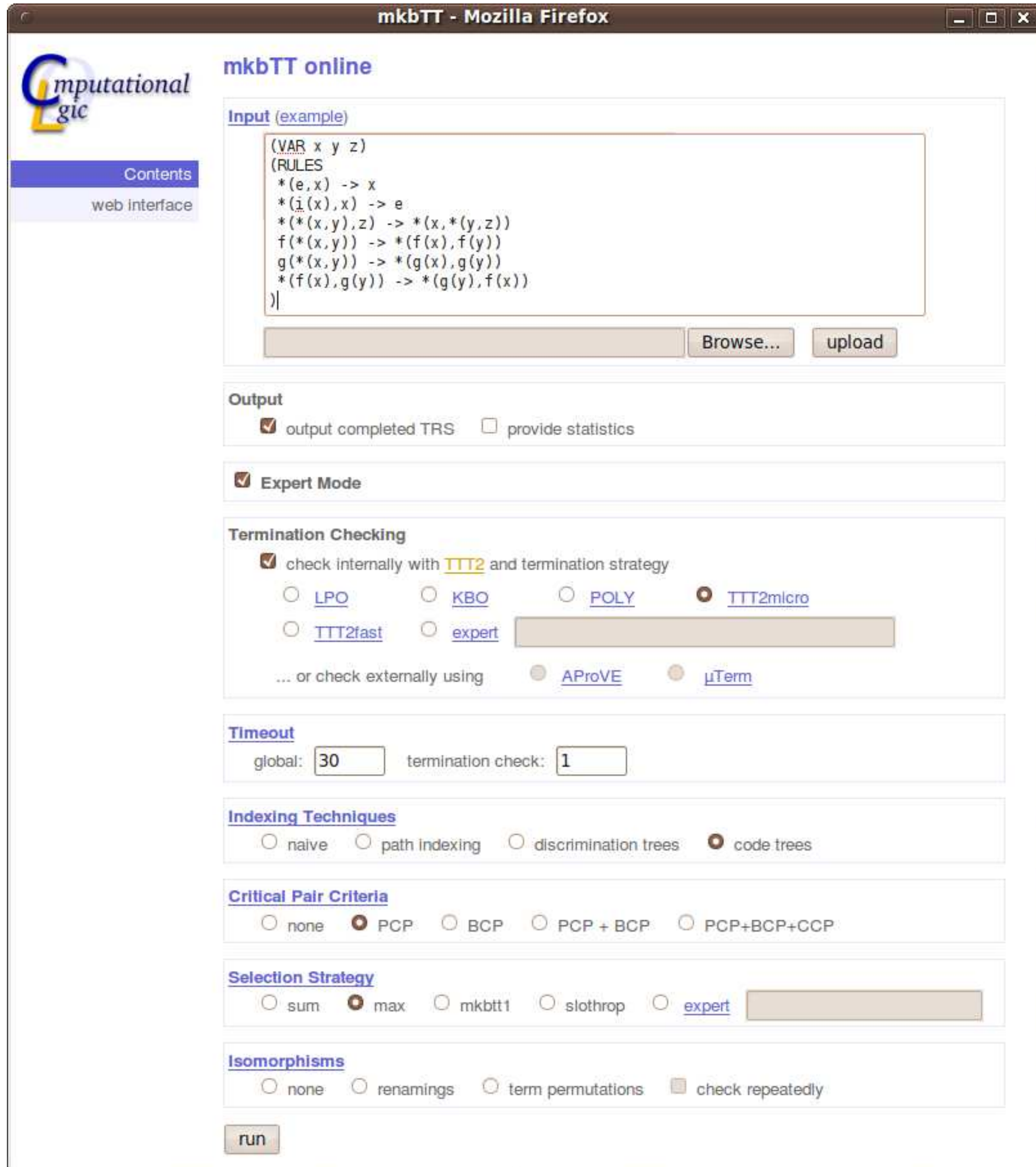


Figure 1: Web interface of $\mathsf{mkb}_{\mathrm{TT}}$.

The current version of $\mathsf{mkb}_{\mathrm{TT}}$ can also produce a canonical presentation for the proof reduction system presented in [Weh05]. Using `ttt2micro` as termination strategy and s_{sum} to control node selections, a complete system is obtained in 115 seconds. According to [Weh05] Waldmeister [Löc02] produces a ground-complete system for this theory. Since the resulting system is not terminating with one of the reduction orders implemented in Waldmeister, it is not clear whether it can produce a complete system for this theory.

The following paragraphs present experimental results for each of the optimizations presented in the previous sections. In all of the following examples, $\mathsf{mkb}_{\mathrm{TT}}$ performs termination checks by interfacing $\mathsf{T_TT_2}$ internally, using `ttt2micro` as termination strategy. The

none		rename		rename+		permutations		permutations+	
(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
188	25	188	25	186	21	188	25	188	25

Table 1: Isomorphisms.

naive			path indexing			discrimination trees			code trees		
(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
19.9	387	91	18.9	345	18	16.5	150	5	15.8	106	5

Table 2: Indexing techniques.

global timeout and the timeout for each termination check were set to 600 and 10 seconds, respectively. If not stated otherwise, we used $s_{\mathtt{max}}$ to control node selections, code trees for indexing operations and neither critical pair criteria nor isomorphisms.

Isomorphisms. The results obtained with $\mathsf{mkb}_{\mathrm{TT}}$ using isomorphism checks are given in Table 1. The columns list (1) the average time in seconds and (2) the average number of processes. As witnessed by the CGE examples, renaming isomorphisms can greatly improve performance on systems with symmetries. Also when tested on the whole database, repeatedly checking for renamings (rename+) pays off slightly. Apparently the advantage gained on systems with isomorphic processes outweighs the overhead required for the remaining input problems. Argument permutations, on the other hand, have little effect. For example, when completing the systems LS94_G0 and GRP012-4th coming from group theory, the number of processes drops from 8 to 4 and 4 to 2 using argument permutations. That also affects the time required for their completion. But since these examples are small and easily completable in any case, the overall performance is not improved.

Indexing Techniques. Table 2 presents results obtained with $\mathsf{mkb}_{\mathrm{TT}}$ using different indexing techniques for rewriting. The columns list (1) the average time to complete a system, (2) the time required for encompassment, and (3) the time for variant retrieval. Retrieving unifiable terms for the computation of overlaps requires only about 1% of the computation time (using naive search in node sets). While path indexing hardly adds anything, the use of discrimination trees allows to decrease this ratio to 0.3%.

Critical Pair Criteria. Table 3 summarizes the outcome of $\mathsf{mkb}_{\mathrm{TT}}$ applying the critical pair criteria mentioned in Section 4. For the column "all", the criteria PCP, BCP and CCP were combined such that every equation deemed superfluous by some criterion got filtered out. The columns list (1) the time required to complete a system, and (2) the number of critical pairs that were recognized as redundant, both for the successful process and for all processes. In the last two rows, the number of successful completions and the total time for all problems in the test series are given. The prefix BGK94 designates examples originating from [Bün94]. Problem Chr89-A_2 is taken from [Chr89], CGE_3 stems from [Stu06], WS06-1 describes the proof reduction system presented in [Weh05], and GRP463-1 is the theory associated with the respective problem in the unit equality division of TPTP [Sut09].

The use of PCP, BCP and the combination of all criteria increments the number of successes by one. Timewise, there are some input problems which exhibit a speedup with

	none	PCP		BCP		CCP		all	
	(1)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
BGK94-D_8	∞	550.9	28/220	550.2	28/212	∞		549.7	28/224
BGK94-D_{16}	105.9	101.1	19/119	104.3	19/112	106.6	8/41	104.7	20/125
Chr89-A_2	126.8	133.7	70/904	134.5	51/771	168.7	25/199	137.5	75/923
CGE_3	198.7	197.7	16/23	198.3	16/20	197.5	8/11	200.4	18/29
GRP463-1	8.6	5.5	24/237	7.1	24/223	9.5	9/43	6.5	27/257
WS06-1	138.6	139.7	0/0	140.3	0/0	139.3	0/0	138.3	0/0
⋮	⋮	⋮		⋮		⋮		⋮	
successes	70	71		71		70		71	
total time	18867	18814		18815		18935		18822	

Table 3: Critical pair criteria.

critical pair criteria, such as BGK-D_{16} or GRP463-1. For other problems such as Chr89-A_2, many critical pairs were filtered out but few of them were actually of use for the successful process. Comparing the implemented criteria, PCP proved to be both the fastest and the most powerful option. BCP recognizes slightly fewer redundancies, but is still feasible. For CCP, the computational overhead clearly outweighs the advantage of the (rather few) superfluous critical pairs. Also using the combination of all criteria is hardly worth the effort, despite the fact that the largest number of critical pairs is filtered out. Overall, critical pair criteria allow for rather small improvements.

Selection Strategies. Table 4 illustrates the effect of different selection strategies. The strategies $s_{\mathtt{sum}}$, $s_{\mathtt{max}}$, $s_{\mathtt{slothrop}}$ and $s_{\mathtt{mkbtt1}}$ are described in Section 5, where the latter two use a size-age ratio of 0.65. The strategy $s_{\mathtt{mkbtt1max}}$ differs from $s_{\mathtt{mkbtt1}}$ in that `ssum` is replaced by `smax`. The columns list (1) the time required to complete a system and (2) the number of selected nodes, i.e., the number of iterations of the main control loop. In the last two rows the number of successful completions and the average time for these are given. The prefix SK90 designates examples originating from [Ste90].

For many problems, the results depend greatly on the selection strategy used. Some problems like CGE_3 perform better when the node size measure `smax` is used. Using `smax` also improves the average time to complete a problem, although overall $s_{\mathtt{sum}}$ could complete the most input problems. There are also some problems like SK90-3.04 where $s_{\mathtt{slothrop}}$ needs the fewest node selections. However, this strategy tends to take more time as the critical pair computation in its node measure is costly to compute. Altogether, 78 examples could be completed with some strategy.

References

[Ala07] B. Alarcón, R. Gutiérrez, J. Iborra, and S. Lucas. Proving termination of context-sensitive rewriting with MU-TERM. In *Proc. 6th PROLE*, *ENTCS*, vol. 188, pp. 105–115. 2007.

[Bac88] L. Bachmair and N. Dershowitz. Critical pair criteria for completion. *Journal of Symbolic Computation*, 6(1):1–18, 1988.

[Bac94] L. Bachmair and N. Dershowitz. Equational inference, canonical proofs, and proof orderings. *Journal of the ACM*, 41(2):236–276, 1994.

[Bün94] R. Bündgen, M. Göbel, and W. Küchlin. A fine-grained parallel completion procedure. In *Proc. 7th ISSAC*, pp. 269–277. 1994.

	$s_{\mathtt{sum}}$		$s_{\mathtt{max}}$		$s_{\mathtt{mkbtt1}}$		$s_{\mathtt{mkbtt1max}}$		$s_{\mathtt{slothrop}}$	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
Chr89-A_2	77.9	153	149.6	150	∞		∞		∞	
SK90-3.04	74.6	133	1.6	39	37.6	105	2.3	42	2.9	33
SK90-3.27	59.1	68	70.0	46	56.8	58	178.5	86	∞	
BGK94-D_8	303.7	217	90.4	134	∞		71.1	105	591.9	160
BGK94-D_{10}	39.8	126	31.7	102	∞		198.4	171	∞	
BGK94-M_{14}	1.48	34	∞		∞		∞		∞	
TPTP/GRP454-1	87.4	168	2.0	38	14.5	75	∞		8.8	40
TPTP/GRP484-1	252.2	324	∞		∞		∞		∞	
CGE_2	138.2	157	9.0	44	7.6	56	9.9	46	15.8	46
CGE_3	∞		189.6	56	∞		121.3	58	343.9	66
$\vdots$	$\vdots$		$\vdots$		$\vdots$		$\vdots$		$\vdots$	
successes	74		71		66		68		69	
average time	22.2		12.8		23.5		15.8		38.9	

Table 4: Selection strategies.

[Chr89] J. Christian. Fast Knuth-Bendix completion. In *Proc. 3rd RTA*, *LNCS*, vol. 355, pp. 551–555. 1989.

[Gie06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proc. 3rd IJCAR*, *LNAI*, vol. 4130, pp. 281–286. 2006.

[Gra96] P. Graf. *Term Indexing*, *LNAI*, vol. 1053. Springer-Verlag, 1996.

[Kap88] D. Kapur, D.R. Musser, and P. Narendran. Only prime superpositions need be considered in the Knuth-Bendix completion procedure. *Journal of Symbolic Computation*, 6(1):19–36, 1988.

[Kor09] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In *Proc. 20th RTA*, *LNCS*, vol. 5595, pp. 295–304. 2009.

[Küc85] W. Küchlin. A confluence criterion based on the generalised Newman lemma. In *Proc. 2nd EUROCAL*, *LNCS*, vol. 204, pp. 390–399. 1985.

[Kur99] M. Kurihara and H. Kondo. Completion for multiple reduction orderings. *Journal of Automated Reasoning*, 23(1):25–42, 1999.

[Löc02] B. Löchner and T. Hillenbrand. A phytography of Waldmeister. *AI Communications*, 15(2-3):127–133, 2002.

[McC92] W. McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Sat08] H. Sato, S. Winkler, M. Kurihara, and A. Middeldorp. Multi-completion with termination tools (system description). In *Proc. 4th IJCAR*, *LNAI*, vol. 5195, pp. 306–312. 2008.

[Sat09] H. Sato, S. Winkler, M. Kurihara, and A. Middeldorp. Constraint-based multi-completion procedures for term rewriting systems. *IEICE Transactions on Information and Systems*, E92-D(2):220–234, 2009.

[Sek01] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning*, pp. 1853–1964. Elsevier Science Publishers, 2001.

[Ste90] J. Steinbach and U. Kühler. Check your ordering – termination proofs and open problems. Tech. Rep. SR-90-25, Universität Kaiserslautern, 1990.

[Stu06] A. Stump and B. Löchner. Knuth-Bendix completion of theories of commuting group endomorphisms. *Information Processing Letters*, 98(5):195–198, 2006.

[Sut09] G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[Vor95] A. Voronkov. The anatomy of Vampire. *Journal of Automated Reasoning*, 15(2):237–265, 1995.

[Vor01] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction. In *Proc. 1st IJCAR*, *LNCS*, vol. 2083, pp. 13–28. 2001.

[Weh05] I. Wehrman and A. Stump. Mining propositional simplification proofs for small validating clauses. In *Proc. 3rd PDPAR*, *ENTCS*, vol. 144, pp. 79–91. 2005.

[Weh06] I. Wehrman, A. Stump, and E.M. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *Proc. 17th RTA*, *LNCS*, vol. 4098, pp. 287–296. 2006.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 385-400
http://rewriting.loria.fr/rta/

MODULAR COMPLEXITY ANALYSIS VIA RELATIVE COMPLEXITY

HARALD ZANKL AND MARTIN KORP

Institute of Computer Science, University of Innsbruck, Austria
E-mail address: {harald.zankl,martin.korp}@uibk.ac.at

Abstract. In this paper we introduce a modular framework which allows to infer (feasible) upper bounds on the (derivational) complexity of term rewrite systems by combining different criteria. All current investigations to analyze the derivational complexity are based on a single termination proof, possibly preceded by transformations. We prove that the modular framework is strictly more powerful than the conventional setting. Furthermore, the results have been implemented and experiments show significant gains in power.

1. Introduction

Term rewriting is a Turing complete model of computation. As an immediate consequence all interesting properties are undecidable. Nevertheless many powerful techniques have been developed for establishing *termination*. The majority of these techniques have been automated successfully. This development has been stimulated by the international competition of termination tools.[1] Most automated analyzers gain their power from a modular treatment of the rewrite system (typically via the dependency pair framework [2,11,22]).

For terminating rewrite systems Hofbauer and Lautemann [14] consider the length of derivations as a measurement for the complexity of rewrite systems. The resulting notion of *derivational complexity* relates the length of a rewrite sequence to the size of its starting term. Thereby it is, e.g., a suitable metric for the complexity of deciding the word problem for a given confluent and terminating rewrite system (since the decision procedure rewrites terms to normal form). If one regards a rewrite system as a program and wants to estimate the maximal number of computation steps needed to evaluate an expression to a result, then the special shape of the starting terms—a function applied to data which is in normal form—can be taken into account. Hirokawa and Moser [12] identified this special form of complexity and named it *runtime complexity*.

To show (feasible) upper complexity bounds currently few techniques are known. Typically termination criteria are restricted such that complexity bounds can be inferred. The early work by Hofbauer and Lautemann [14] considers polynomial interpretations, suitably

1998 ACM Subject Classification: F.2 Analysis of Algorithms and Problem Complexity, F.4 Mathematical Logic and Formal Languages.

Key words and phrases: term rewriting, complexity analysis, relative complexity, derivation length.

This research is supported by FWF (Austrian Science Fund) project P18763.

[1] http://termcomp.uibk.ac.at

© Harald Zankl and Martin Korp
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.385

restricted, to admit quadratic derivational complexity. Match-bounds [9] and arctic matrix interpretations [16] induce linear derivational complexity and triangular matrix interpretations [19] admit polynomially long derivations (the dimension of the matrices yields the degree of the polynomial). All these methods share the property that until now they have been used directly only, meaning that a single termination technique has to orient all rules in one go. However, using direct criteria exclusively is problematic due to their restricted power.

In [12,13] Hirokawa and Moser lift many aspects of the dependency pair framework from termination analysis into the complexity setting, resulting in the notion of weak dependency pairs. So for the special case of runtime complexity for the first time a modular approach has been introduced. There the modular aspect amounts to using different interpretation based criteria for (parts of the) weak dependency graph and the usable rules. However, still all rewrite rules considered must be oriented strictly in one go and only restrictive criteria may be applied for the usable rules. A further drawback of weak dependency pairs is that they may only be used for bounding runtime complexity while there seems to be no hope to generalize the method to derivational complexity.

In this paper we present a different approach which admits a fully modular treatment. The approach is general enough that it applies to derivational complexity (and hence also to runtime complexity) and basic enough that it allows to combine completely different complexity criteria such as match-bounds and triangular matrix interpretations. By the modular combination of different criteria also gains in power are achieved. These gains come in two flavors. On one hand our approach allows to obtain lower complexity bounds for several rewrite systems where bounds have already been established before and on the other hand we found bounds for systems that could not be dealt with so far automatically.

The remainder of the paper is organized as follows. In Section 2 preliminaries about term rewriting and complexity analysis are fixed. Afterwards, Section 3 familiarizes the reader with the concept of a suitable complexity measurement for relative rewriting. Section 4 formulates a modular framework for complexity analysis based on relative complexity. In Section 5 we show that the modular setting is strictly more powerful than the conventional approach. Our results have been implemented in the complexity prover $\mathsf{C_\partial T}$. The technical details can be inferred from Section 6. Section 7 is devoted to demonstrate the power of the modular treatment by means of an empirical evaluation. Section 8 concludes.

2. Preliminaries

We assume familiarity with (relative) term rewriting [3, 10, 21]. Let $\mathcal{F}$ be a signature and $\mathcal{V}$ be a disjoint set of variables. By $\mathcal{T}(\mathcal{F}, \mathcal{V})$ we denote the set of terms over $\mathcal{F}$ and $\mathcal{V}$ and by $\mathcal{T}(\mathcal{F})$ the set of ground terms over $\mathcal{F}$. We write $\mathcal{F}\mathsf{un}(t)$ for the set of function symbols occurring in a term t. The size of a term t is denoted $|t|$ and $\|t\|$ computes the number of occurrences of function symbols in t. Positions are used to address symbol occurrences in terms. Given a term t and a position $p \in \mathcal{P}\mathsf{os}(t)$, we write $t(p)$ for the symbol at position p. We use $\mathcal{F}\mathcal{P}\mathsf{os}(t)$ to denote the subset of positions $p \in \mathcal{P}\mathsf{os}(t)$ such that $t(p) \in \mathcal{F}$.

A *rewrite rule* is a pair of terms (l, r), written $l \to r$ such that l is not a variable and all variables in r are contained in l. A *term rewrite system* (TRS for short) is a set of rewrite rules. For complexity analysis we assume TRSs to be finite. A TRS $\mathcal{R}$ is said to be *duplicating* if there exists a rewrite rule $l \to r \in \mathcal{R}$ and a variable x that occurs more often in r than in l. A TRS is called *linear* if for all rewrite rules $l \to r \in \mathcal{R}$ any variable x

occurs at most once in l and r, respectively. A *rewrite relation* is a binary relation on terms that is closed under contexts and substitutions. For a TRS $\mathcal{R}$ we define $\to_{\mathcal{R}}$ to be the smallest rewrite relation that contains $\mathcal{R}$. As usual $\to^*$ denotes the reflexive and transitive closure of $\to$ and $\to^n$ the n-th iterate of $\to$. A *relative* TRS $\mathcal{R}/\mathcal{S}$ is a pair of TRSs $\mathcal{R}$ and $\mathcal{S}$ with the induced rewrite relation $\to_{\mathcal{R}/\mathcal{S}} = \to_{\mathcal{S}}^* \cdot \to_{\mathcal{R}} \cdot \to_{\mathcal{S}}^*$. In the sequel we will sometimes identify a TRS $\mathcal{R}$ with the relative TRS $\mathcal{R}/\varnothing$.

The *derivation length* of a term t with respect to a relation $\to$ and a set of terms L is defined as follows: $\mathsf{dl}(t, \to, L) = \max\{m \mid \exists u\; t \to^m u,\, t \in L\}$. We abbreviate $\mathsf{dl}(t, \to, \mathcal{T}(\mathcal{F}, \mathcal{V}))$ by $\mathsf{dl}(t, \to)$. The *derivational complexity* computes the maximal derivation length of all terms up to a given size, i.e., $\mathsf{dc}(n, \to) = \max\{\mathsf{dl}(t, \to) \mid |t| \leqslant n\}$. Sometimes we say that $\mathcal{R}$ ($\mathcal{R}/\mathcal{S}$) has linear, quadratic, etc. derivational complexity if $\mathsf{dc}(n, \to_{\mathcal{R}})$ ($\mathsf{dc}(n, \to_{\mathcal{R}/\mathcal{S}})$) can be bounded by a linear, quadratic, etc. polynomial in n.

2.1. Triangular Matrix Interpretations

An $\mathcal{F}$*-algebra* $\mathcal{A}$ consists of a non-empty carrier A and a set of interpretations $f_{\mathcal{A}}$ for every $f \in \mathcal{F}$. By $[\alpha](\cdot)_{\mathcal{A}}$ we denote the usual evaluation function of $\mathcal{A}$ according to an assignment α. An $\mathcal{F}$-algebra $\mathcal{A}$ together with two relations $\succ$ and $\succeq$ on A is called a *monotone algebra* if every $f_{\mathcal{A}}$ is monotone with respect to $\succ$ and $\succeq$, $\succ$ is a well-founded order, and $\succ \cdot \succeq \subseteq \succ$ and $\succeq \cdot \succ \subseteq \succ$ holds. Any monotone algebra $(\mathcal{A}, \succ, \succeq)$ induces a well-founded order on terms, i.e., $s \succ_{\mathcal{A}} t$ if for any assignment α the condition $[\alpha](s)_{\mathcal{A}} \succ [\alpha](t)_{\mathcal{A}}$ holds. The order $\succeq_{\mathcal{A}}$ is defined similarly. A relative TRS $\mathcal{R}/\mathcal{S}$ is *compatible* with a monotone algebra $(\mathcal{A}, \succ_{\mathcal{A}}, \succeq_{\mathcal{A}})$ if $\mathcal{R} \subseteq \succ_{\mathcal{A}}$ and $\mathcal{S} \subseteq \succeq_{\mathcal{A}}$. *Matrix interpretations* $(\mathcal{M}, \succ_{\mathcal{M}}, \succeq_{\mathcal{M}})$ (often just denoted $\mathcal{M}$) are a special form of monotone algebras. Here the carrier is $\mathbb{N}^d$ for some fixed dimension $d \in \mathbb{N} \setminus \{0\}$. The order $\succeq_{\mathcal{M}}$ is the point-wise extension of $\geqslant_{\mathbb{N}}$ to vectors and $\vec{u} \succ_{\mathcal{M}} \vec{v}$ if $\vec{u}_1 >_{\mathbb{N}} \vec{v}_1$ and $\vec{u} \succeq_{\mathcal{M}} \vec{v}$. If every $f \in \mathcal{F}$ of arity n is interpreted as $f_{\mathcal{M}}(\vec{x_1}, \ldots, \vec{x_n}) = F_1\vec{x_1} + \cdots + F_n\vec{x_n} + \vec{f}$ where $F_i \in \mathbb{N}^{d \times d}$ and $\vec{f} \in \mathbb{N}^d$ for all $1 \leqslant i \leqslant n$ then monotonicity of $\succ_{\mathcal{M}}$ is achieved by demanding $F_{i(1,1)} \geqslant 1$ for any $1 \leqslant i \leqslant n$. A matrix interpretation where for every $f \in \mathcal{F}$ all F_i ($1 \leqslant i \leqslant \mathsf{arity}(f)$) are upper triangular is called *triangular matrix interpretation* (abbreviated by TMI). A square matrix A of dimension d is of *upper triangular* shape if $A_{(i,i)} \leqslant 1$ and $A_{(i,j)} = 0$ if $i > j$ for all $1 \leqslant i, j \leqslant d$. For historic reasons a TMI based on matrices of dimension one is also called *strongly linear interpretation* (SLI for short). In [19] it is shown that the derivational complexity of a TRS $\mathcal{R}$ is bounded by a polynomial of degree d if there exists a TMI $\mathcal{M}$ of dimension d such that $\mathcal{R}$ is compatible with $\mathcal{M}$.

2.2. Match-Bounds

Let $\mathcal{F}$ be a signature, $\mathcal{R}$ a TRS over $\mathcal{F}$, and $L \subseteq \mathcal{T}(\mathcal{F})$ a set of ground terms. The set $\{t \in \mathcal{T}(\mathcal{F}) \mid s \to_{\mathcal{R}}^* t \text{ for some } s \in L\}$ of reducts of L is denoted by $\to_{\mathcal{R}}^*(L)$. Given a set $N \subseteq \mathbb{N}$ of natural numbers, the signature $\mathcal{F} \times N$ is denoted by $\mathcal{F}_N$. Here function symbols (f, n) with $f \in \mathcal{F}$ and $n \in N$ have the same arity as f and are written as f_n. The mappings $\mathsf{lift}_c\colon \mathcal{F} \to \mathcal{F}_{\mathbb{N}}$, $\mathsf{base}\colon \mathcal{F}_{\mathbb{N}} \to \mathcal{F}$, and $\mathsf{height}\colon \mathcal{F}_{\mathbb{N}} \to \mathbb{N}$ are defined as $\mathsf{lift}_c(f) = f_c$, $\mathsf{base}(f_c) = f$, and $\mathsf{height}(f_c) = c$ for all $f \in \mathcal{F}$ and $c \in \mathbb{N}$. They are extended to terms, sets of terms, and TRSs in the obvious way. The TRS $\mathsf{match}(\mathcal{R})$ over the signature $\mathcal{F}_{\mathbb{N}}$ consists of all rewrite rules $l' \to \mathsf{lift}_c(r)$ for which there exists a rule $l \to r \in \mathcal{R}$ such that $\mathsf{base}(l') = l$ and $c = 1 + \min\{\mathsf{height}(l'(p)) \mid p \in \mathcal{FP}\mathsf{os}(l)\}$. Here $c \in \mathbb{N}$. The restriction of

$\mathsf{match}(\mathcal{R})$ to the signature $\mathcal{F}_{\{0,\ldots,c\}}$ is denoted by $\mathsf{match}_c(\mathcal{R})$. Let L be a set of terms. The TRS $\mathcal{R}$ is called *match-bounded* for L if there exists a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms in $\to^*_{\mathsf{match}(\mathcal{R})}(\mathsf{lift}_0(L))$ is at most c. If we want to make the bound c precise, we say that $\mathcal{R}$ is match-bounded for L *by* c. If we do not specify the set of terms L then it is assumed that $L = \mathcal{T}(\mathcal{F})$. If a linear TRS $\mathcal{R}$ is match-bounded for a language L then $\mathcal{R}$ is terminating on L. Furthermore $\mathsf{dl}(t, \to_{\mathcal{R}}, L)$ is bounded by a linear polynomial for any term $t \in L$ [9].

In order to prove that a TRS $\mathcal{R}$ is match-bounded for some language L, the idea is to construct a tree automaton that is compatible with $\mathsf{match}(\mathcal{R})$ and $\mathsf{lift}_0(L)$. A tree automaton $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ is said to be *compatible* with some TRS $\mathcal{R}$ and some language L if $L \subseteq \mathcal{L}(\mathcal{A})$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma \colon \mathcal{V}\mathsf{ar}(l) \to Q$ such that $l\sigma \to^*_\Delta q$ it holds that $r\sigma \to^*_\Delta q$.

3. Relative Complexity

In this section we introduce complexity analysis for relative rewriting, i.e., given $\mathcal{R}/\mathcal{S}$ only the $\mathcal{R}$-steps contribute to the complexity. To estimate the derivational complexity of a relative TRS $\mathcal{R}/\mathcal{S}$, a pair of orderings $(\succ, \succeq)$ will be used such that $\mathcal{R} \subseteq \succ$ and $\mathcal{S} \subseteq \succeq$. The necessary properties of these orderings are given in the next definition.

Definition 3.1. A *complexity pair* $(\succ, \succeq)$ consists of two finitely branching rewrite relations $\succ$ and $\succeq$ that are *compatible*, i.e., $\succeq \cdot \succ \subseteq \succ$ and $\succ \cdot \succeq \subseteq \succ$. We call a relative TRS $\mathcal{R}/\mathcal{S}$ *compatible* with a complexity pair $(\succ, \succeq)$ if $\mathcal{R} \subseteq \succ$ and $\mathcal{S} \subseteq \succeq$.

The next lemma states that given a complexity pair $(\succ, \succeq)$ and a compatible relative TRS $\mathcal{R}/\mathcal{S}$, the $\succ$ ordering is crucial for estimating the derivational complexity of $\mathcal{R}/\mathcal{S}$. Intuitively the result states that every $\mathcal{R}/\mathcal{S}$-step gives rise to at least one $\succ$-step.

Lemma 3.2. *Let $(\succ, \succeq)$ be a complexity pair and $\mathcal{R}/\mathcal{S}$ be a compatible relative TRS. Then for any term t terminating on $\mathcal{R}/\mathcal{S}$ we have $\mathsf{dl}(t, \succ) \geqslant \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$.*

Proof. By assumption $\mathcal{R}/\mathcal{S}$ is compatible with $(\succ, \succeq)$. Since $\succ$ and $\succeq$ are rewrite relations $\to_{\mathcal{R}} \subseteq \succ$ and $\to_{\mathcal{S}} \subseteq \succeq$ holds. From the compatibility of $\succ$ and $\succeq$ we obtain $\to_{\mathcal{R}/\mathcal{S}} \subseteq \succ$. Hence for any sequence

$$t \to_{\mathcal{R}/\mathcal{S}} t_1 \to_{\mathcal{R}/\mathcal{S}} t_2 \to_{\mathcal{R}/\mathcal{S}} \cdots$$

also

$$t \quad \succ \quad t_1 \quad \succ \quad t_2 \quad \succ \quad \cdots$$

holds. The result follows immediately from this. ∎

Obviously $\succ$ must be at least well-founded if *finite* complexities should be estimated. Because we are especially interested in feasible upper bounds the following corollary is specialized to polynomials.

Corollary 3.3. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS compatible with a complexity pair $(\succ, \succeq)$. If the derivational complexity of $\succ$ is linear, quadratic, etc. then the derivational complexity of $\mathcal{R}/\mathcal{S}$ is linear, quadratic, etc.*

Proof. By Lemma 3.2. ∎

This corollary allows to investigate the derivational complexity of (compatible) complexity pairs instead of the derivational complexity of the underlying relative TRS. Complexity pairs can, e.g., be obtained by TMIs.

Theorem 3.4. *Let* $(\mathcal{M}, \succ_{\mathcal{M}}, \succeq_{\mathcal{M}})$ *be a TMI of dimension* d*. Then* $(\succ_{\mathcal{M}}, \succeq_{\mathcal{M}})$ *is a complexity pair. Furthermore* $\mathsf{dl}(t, \succ_{\mathcal{M}})$ *is bounded by a polynomial of degree* d *for any term* t*.*

Proof. Straightforward from [19, Theorem 6]. ∎

The following example familiarizes the reader with relative derivational complexity analysis.

Example 3.5. Consider the relative TRS $\mathcal{R}/\mathcal{S}$ where $\mathcal{R} = \{\mathsf{f}(x) \to x\}$ and $\mathcal{S} = \{\mathsf{a} \to \mathsf{a}\}$. Then the SLI $\mathcal{M}$ satisfying $f_{\mathcal{M}}(x) = x + 1$ and $\mathsf{a}_{\mathcal{M}} = 0$ induces the complexity pair $(\succ_{\mathcal{M}}, \succeq_{\mathcal{M}})$. Furthermore $\mathcal{R}/\mathcal{S}$ is compatible with $(\succ_{\mathcal{M}}, \succeq_{\mathcal{M}})$. Theorem 3.4 gives a linear bound on $\succ_{\mathcal{M}}$. Hence $\mathcal{R}/\mathcal{S}$ admits (at most) linear derivational complexity by Corollary 3.3. It is easy to see that this bound is tight.

4. Modular Complexity Analysis

Although Theorem 3.4 in combination with Corollary 3.3 is already powerful, a severe drawback is that given a relative TRS $\mathcal{R}/\mathcal{S}$ all rules in $\mathcal{R}$ must be strictly oriented by $\succ_{\mathcal{M}}$. The next (abstract) example demonstrates the basic idea of an approach that allows to weaken this burden.

Example 4.1. Consider the relative TRS $\mathcal{R}/\mathcal{S}$ where $\mathcal{R} = \{r, r'\}$. Instead of estimating $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$ directly, we want to bound it by $\mathsf{dl}(t, \to_{\{r\}/(\{r'\} \cup \mathcal{S})}) + \mathsf{dl}(t, \to_{\{r'\}/(\{r\} \cup \mathcal{S})})$. To proceed we separate the r from the r'-steps. I.e., any reduction sequence in $\mathcal{R}/\mathcal{S}$ can be written as

$$t \quad \to_{\{r\}/\mathcal{S}} \quad t_1 \quad \to_{\{r\}/\mathcal{S}} \quad t_2 \quad \to_{\{r'\}/\mathcal{S}} \quad t_3 \quad \to_{\{r\}/\mathcal{S}} \quad \cdots$$

which immediately gives rise to a sequence

$$t \to_{\{r\}/(\{r'\} \cup \mathcal{S})} t_1 \to_{\{r\}/(\{r'\} \cup \mathcal{S})} t_2 \to_{\{r'\}/(\{r\} \cup \mathcal{S})} t_3 \to_{\{r\}/(\{r'\} \cup \mathcal{S})} \cdots$$

Obviously $\mathsf{dl}(t, \to_{\{r\}/(\{r'\} \cup \mathcal{S})}) + \mathsf{dl}(t, \to_{\{r'\}/(\{r\} \cup \mathcal{S})}) \geqslant \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$.

Next we formalize the main observation needed for modular complexity analysis of a relative TRS $\mathcal{R}/\mathcal{S}$. As already indicated in the example above the key idea is that every rule from $\mathcal{R}$ can be investigated relative to the other rules in $\mathcal{R}$ and $\mathcal{S}$. The next theorem states the main result in this direction. In the sequel we assume that $\mathcal{R}_i \subseteq \mathcal{R}$ for any i. Furthermore, sometimes $\mathcal{R}_i$ refers to $\{r_i\}$ if $\mathcal{R} = \{r_1, \ldots, r_n\}$ and $1 \leqslant i \leqslant n$. The context clarifies if $\mathcal{R}_i$ is an arbitrary subset of $\mathcal{R}$ or its i-th rule. In addition we denote by $\mathcal{S}_i$ the TRS $(\mathcal{S} \cup \mathcal{R}) \setminus \mathcal{R}_i$. If $\mathcal{S}$ is not explicitly mentioned then $\mathcal{S} = \varnothing$.

Theorem 4.2. *Let* $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$ *be a relative TRS for which the term* t *terminates. Then* $\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}) \geqslant \mathsf{dl}(t, \to_{(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}})$.

Proof. We abbreviate $\mathcal{R}_1 \cup \mathcal{R}_2$ by $\mathcal{R}$. Assume that $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}}) = m$. Then there exists a rewrite sequence

$$t \to_{\mathcal{R}/\mathcal{S}} t_1 \to_{\mathcal{R}/\mathcal{S}} t_2 \to_{\mathcal{R}/\mathcal{S}} \cdots \to_{\mathcal{R}/\mathcal{S}} t_{m-1} \to_{\mathcal{R}/\mathcal{S}} t_m \tag{1}$$

of length m. Next we investigate this sequence for every relative TRS $\mathcal{R}_i/\mathcal{S}_i$ ($1 \leqslant i \leqslant 2$) where m_i overestimates how often rules from $\mathcal{R}_i$ have been applied in the original sequence. Fix i. If the sequence (1) does not contain an $\mathcal{R}_i$ step then $t \to^m_{\mathcal{S}_i} t_m$ and $m_i = 0$. In the other case there exists a maximal (with respect to m_i) sequence

$$t \to_{\mathcal{R}_i/\mathcal{S}_i} t_{i_1} \to_{\mathcal{R}_i/\mathcal{S}_i} t_{i_2} \to_{\mathcal{R}_i/\mathcal{S}_i} \cdots \to_{\mathcal{R}_i/\mathcal{S}_i} t_{i_{m_i-1}} \to_{\mathcal{R}_i/\mathcal{S}_i} t_m \tag{2}$$

where $i_1 < i_2 < \cdots < i_{m_i-1} < m$. Together with the fact that every rewrite rule in $\mathcal{R}$ is either contained in $\mathcal{R}_1$ or $\mathcal{R}_2$ we have $m_1 + m_2 \geqslant m$. If $m_i = 0$ we obviously have $\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i}) \geqslant m_i$ and if $t \to^{m_i}_{\mathcal{R}_i/\mathcal{S}_i} t_m$ with $m_i > 0$ we know that $\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i}) \geqslant m_i$ by the choice of sequence (2). (Note that in both cases it can happen that $\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i}) > m_i$ because sequence (1) need not be maximal with respect to $\to_{\mathcal{R}_i/\mathcal{S}_i}$.) Putting things together yields

$$\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}) \geqslant m_1 + m_2 \geqslant m = \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$$

which concludes the proof. ∎

As already indicated, the reverse direction of the above theorem does not hold. This is illustrated by the following example.

Example 4.3. Consider the relative TRS $\mathcal{R}/\mathcal{S}$ with $\mathcal{R} = \{\mathsf{a} \to \mathsf{b}, \mathsf{a} \to \mathsf{c}\}$ and $\mathcal{S} = \varnothing$. We have $\mathsf{a} \to_{\mathcal{R}/\mathcal{S}} \mathsf{b}$ or $\mathsf{a} \to_{\mathcal{R}/\mathcal{S}} \mathsf{c}$. Hence $\mathsf{dl}(\mathsf{a}, \to_{\mathcal{R}/\mathcal{S}}) = 1$. However, the sum of the derivation lengths $\mathsf{dl}(\mathsf{a}, \to_{\mathcal{R}_1/\mathcal{S}_1})$ and $\mathsf{dl}(\mathsf{a}, \to_{\mathcal{R}_2/\mathcal{S}_2})$ is 2.

Although for Theorem 4.2 equality cannot be established the next result states that for complexity analysis this does not matter.

Theorem 4.4. *Let $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$ be a relative TRS for which the term t terminates. Then* $\mathsf{max}\{\mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1})), \mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}))\} = \mathcal{O}(\mathsf{dl}(t, \to_{(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}}))$.

Proof. We abbreviate the union $\mathcal{R}_1 \cup \mathcal{R}_2$ by $\mathcal{R}$. The $\geqslant$-direction follows immediately from Theorem 4.2 and some basic properties of the $\mathcal{O}$-notation like $\mathsf{max}_{1 \leqslant i \leqslant 2}\{\mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i}))\} = \mathcal{O}(\sum_{1 \leqslant i \leqslant 2}\{\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i})\})$. For the $\leqslant$-direction we select a $j \in \{1, 2\}$ such that $\mathcal{R}_j/\mathcal{S}_j$ determines the complexity of $\mathcal{R}/\mathcal{S}$, i.e., $\mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_j/\mathcal{S}_j})) = \mathsf{max}_{1 \leqslant i \leqslant 2}\{\mathcal{O}(\mathsf{dl}(t, \to_{\mathcal{R}_i/\mathcal{S}_i}))\}$. By construction of $\mathcal{R}_j/\mathcal{S}_j$ any derivation of the form

$$t \to_{\mathcal{R}_j/\mathcal{S}_j} t_1 \to_{\mathcal{R}_j/\mathcal{S}_j} \cdots \to_{\mathcal{R}_j/\mathcal{S}_j} t_m$$

induces a derivation

$$t \to^+_{\mathcal{R}/\mathcal{S}} t_1 \to^+_{\mathcal{R}/\mathcal{S}} \cdots \to^+_{\mathcal{R}/\mathcal{S}} t_m$$

of at least the same length. Putting things together finishes the proof. ∎

Theorems 4.2 and 4.4 allow to split a relative TRS $\mathcal{R}/\mathcal{S}$ into *smaller* components $\mathcal{R}_1/\mathcal{S}_1$ and $\mathcal{R}_2/\mathcal{S}_2$—such that $\mathcal{R}_1 \cup \mathcal{R}_2 = \mathcal{R}$—and evaluate the complexities of these components (e.g., by different complexity pairs) independently. Note that this approach is not restricted to relative rewriting. To estimate the complexity of a (non-relative) TRS $\mathcal{R}$ just consider the relative TRS $\mathcal{R}/\varnothing$.

In Theorem 3.4 we have already seen one method to obtain complexity pairs. In the next subsection we study how the match-bounds technique can be suited for relative complexity analysis. Afterwards, in Section 4.2, we show that under certain circumstances the derivational complexity of a relative TRS $\mathcal{R}/\mathcal{S}$ can be estimated by considering $\mathcal{R}_1/\mathcal{S}_1$ and $\mathcal{R}_2/\mathcal{S}$. Note that Theorem 4.2 requires $\mathcal{R}_2/\mathcal{S}_2$ with $\mathcal{S}_2 = \mathcal{R}_1 \cup \mathcal{S}$ instead of $\mathcal{R}_2/\mathcal{S}$.

4.1. Complexity Pairs via Match-Bounds

It is easy to use the match-bound technique for estimating the derivational complexity of a relative TRS $\mathcal{R}/\mathcal{S}$; just check for match-boundedness of $\mathcal{R} \cup \mathcal{S}$. This process either succeeds by proving that the combined TRS is match-bounded, or, when $\mathcal{R} \cup \mathcal{S}$ cannot be proved to be match-bounded, it fails. Since the construction of a compatible tree automaton does not terminate for TRSs that are not match-bounded, the latter situation typically does not happen. This behavior causes two serious problems. On the one hand we cannot benefit from Theorem 4.4 because whenever we split a relative TRS $\mathcal{R}/\mathcal{S}$ into smaller components $\mathcal{R}_1/\mathcal{S}_1$ and $\mathcal{R}_2/\mathcal{S}_2$ then $\mathcal{R}_1 \cup \mathcal{S}_1$ is match-bounded if and only if $\mathcal{R}_2 \cup \mathcal{S}_2$ is match-bounded since both TRSs coincide. On the other hand the match-bound technique cannot cooperate with other techniques since either linear derivational complexity of all or none of the rules in $\mathcal{R}$ is shown. In [23] these problems have been addressed by checking if there is a $c \in \mathbb{N}$ such that the maximum height of function symbols occurring in terms in derivations caused by the TRS $\mathsf{match}_{c+1}(\mathcal{R}) \cup \mathsf{match}_c(\mathcal{S}) \cup \mathsf{lift}_c(\mathcal{S})$ is at most c. Below we follow a different approach which seems to be more suitable for our setting. That is, given some relative TRS $\mathcal{R}/\mathcal{S}$ we adapt the match-bound technique in such a way that it can orient the rewrite rules in $\mathcal{R}$ strictly and the ones in $\mathcal{S}$ weakly. To this end we introduce a new enrichment $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$. The key idea behind the new enrichment is to keep heights until a labeled version of a rule in $\mathcal{R}$ is applied.

To simplify the presentation we consider linear TRSs only. The extension to non-duplicating TRSs is straightforward and follows [17].

Definition 4.5. Let $\mathcal{S}$ be a TRS over a signature $\mathcal{F}$. The TRS $\mathsf{match\text{-}RT}(\mathcal{S})$ over the signature $\mathcal{F}_\mathbb{N}$ consists of all rules $l' \to \mathsf{lift}_c(r)$ such that $\mathsf{base}(l') \to r \in \mathcal{S}$. Here $c = \mathsf{height}(l'(\epsilon))$ if $\|\mathsf{base}(l')\| \geqslant \|r\|$ and $\mathsf{lift}_c(\mathsf{base}(l')) = l'$, and $c = \min\{1 + \mathsf{height}(l'(p)) \mid p \in \mathcal{FP}\mathsf{os}(l')\}$ otherwise. Given a relative TRS $\mathcal{R}/\mathcal{S}$, the relative TRS $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ is defined as $\mathsf{match}(\mathcal{R})/\mathsf{match\text{-}RT}(\mathcal{S})$.

To satisfy the requirement that the new enrichment $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ counts only $\mathcal{R}$-steps we try to keep the labels of the function symbols in a contracted redex if a size-preserving or size-decreasing rewrite rule in $\mathcal{S}$ is applied. We need this requirement on the size that the new enrichment is compatible with multi-set orderings $\succ_{\mathsf{mul}}$ and $\succeq_{\mathsf{mul}}$ which induce a linear complexity.

Example 4.6. Consider the TRS $\mathcal{R} = \{\mathsf{rev}(x) \to \mathsf{rev}'(x, \mathsf{nil}), \mathsf{rev}'(\mathsf{nil}, y) \to y\}$ and the TRS $\mathcal{S} = \{\mathsf{rev}'(\mathsf{cons}(x, y), z) \to \mathsf{rev}'(y, \mathsf{cons}(x, z))\}$. Then $\mathsf{match}(\mathcal{R})$ contains the rules

$$\begin{array}{ll} \mathsf{rev}_0(x) \to \mathsf{rev}'_1(x, \mathsf{nil}_1) & \mathsf{rev}'_0(\mathsf{nil}_0, y) \to y \\ \mathsf{rev}_1(x) \to \mathsf{rev}'_2(x, \mathsf{nil}_2) & \mathsf{rev}'_0(\mathsf{nil}_1, y) \to y \\ \cdots & \cdots \end{array}$$

and $\mathsf{match\text{-}RT}(\mathcal{S})$ contains

$$\begin{array}{c} \mathsf{rev}'_0(\mathsf{cons}_0(x, y), z) \to \mathsf{rev}'_0(y, \mathsf{cons}_0(x, z)) \\ \mathsf{rev}'_0(\mathsf{cons}_1(x, y), z) \to \mathsf{rev}'_1(y, \mathsf{cons}_1(x, z)) \\ \cdots \end{array}$$

Both infinite TRSs together constitute $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$.

In order to prove that a relative TRS $\mathcal{R}/\mathcal{S}$ admits at most linear derivational complexity using the enrichment $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ we need the property defined below.

Definition 4.7. Let $\mathcal{R}/\mathcal{S}$ be a relative TRS. We call $\mathcal{R}/\mathcal{S}$ *match-RT-bounded* for a set of terms L if there exists a number $c \in \mathbb{N}$ such that the height of function symbols occurring in terms in $\rightarrow^*_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})}(\mathsf{lift}_0(L))$ is at most c.

Let $\mathcal{M}\mathcal{F}\mathsf{un}(t) = \{\mathsf{height}(f) \mid f \in \mathcal{F}\mathsf{un}(t)\}$ denote the multiset of the heights of the function symbols that occur in the term t. The precedence $\succ$ on $\mathbb{N}$ is defined as $i \succ j$ if and only if $j >_{\mathbb{N}} i$. Let $\mathcal{M}\mathsf{ul}(\mathbb{N})$ denote all multisets over $\mathbb{N}$. We write $s \succ_{\mathsf{mul}} t$ for terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ if $\mathcal{M}\mathcal{F}\mathsf{un}(s)$ is greater than $\mathcal{M}\mathcal{F}\mathsf{un}(t)$ in the multiset extension of $\succ$ on $\mathbb{N}$. Similarly, $s \succeq_{\mathsf{mul}} t$ if $s \succ_{\mathsf{mul}} t$ or $\mathcal{M}\mathcal{F}\mathsf{un}(s) = \mathcal{M}\mathcal{F}\mathsf{un}(t)$.

A key property of the relative TRS $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ is that the rewrite rules which originate from rules in $\mathcal{R}$ can be oriented by $\succ_{\mathsf{mul}}$ and all other rules can be oriented by $\succeq_{\mathsf{mul}}$. Hence the derivation length induced by the relative TRS $\mathcal{R}/\mathcal{S}$ on some language L is bounded by a linear polynomial whenever $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for L.

Lemma 4.8. *Let $\mathcal{R}$ and $\mathcal{S}$ be two non-duplicating TRSs. We have $\rightarrow_{\mathsf{match}(\mathcal{R})} \subseteq \succ_{\mathsf{mul}}$ and $\rightarrow_{\mathsf{match\text{-}RT}(\mathcal{S})} \subseteq \succeq_{\mathsf{mul}}$.*

Proof. From the proof of [9, Lemma 17] we know that for a non-duplicating TRS $\mathcal{R}$ and two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \succ_{\mathsf{mul}} t$ whenever $s \rightarrow_{\mathsf{match}(\mathcal{R})} t$. If $s \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{S})} t$ then either $\mathcal{M}\mathcal{F}\mathsf{un}(s) \supseteq \mathcal{M}\mathcal{F}\mathsf{un}(t)$ or $s \rightarrow_{\mathsf{match}(\mathcal{S})} t$ by Definition 4.5. In the former case we have $s \succeq_{\mathsf{mul}} t$ and in the latter one $s \succ_{\mathsf{mul}} t$ and hence $s \succeq_{\mathsf{mul}} t$ by the definition of $\succeq_{\mathsf{mul}}$. This concludes the proof of the lemma. ∎

Theorem 4.9. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS such that $\mathcal{R}$ and $\mathcal{S}$ are linear and let L be a set of terms. If $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for L then $\mathcal{R}/\mathcal{S}$ is terminating on L. Furthermore $\mathsf{dl}(t, \rightarrow_{\mathcal{R}/\mathcal{S}}, L)$ is bounded by a linear polynomial for any term $t \in L$.*

Proof. At first we show that $\mathcal{R}/\mathcal{S}$ is terminating on L. Assume to the contrary that there is an infinite rewrite sequence of the form

$$t_1 \rightarrow_{\mathcal{R}/\mathcal{S}} t_2 \rightarrow_{\mathcal{R}/\mathcal{S}} t_3 \rightarrow_{\mathcal{R}/\mathcal{S}} \cdots$$

with $t_1 \in L$. Because $\mathcal{R}$ and $\mathcal{S}$ are left-linear, this derivation can be lifted to an infinite $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ rewrite sequence

$$t'_1 \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})} t'_2 \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})} t'_3 \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})} \cdots$$

starting from $t'_1 = \mathsf{lift}_0(t_1)$. Since the original sequence contains infinitely many $\mathcal{R}$-steps the lifted sequence contains infinitely many $\mathsf{match}(\mathcal{R})$-steps. Moreover, because $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for L, there is a $c \geqslant 0$ such that the height of every function symbol occurring in a term in the lifted sequence is at most c. Lemma 4.8 as well as compatibility of $\succ_{\mathsf{mul}}$ and $\succeq_{\mathsf{mul}}$ yields $t'_i \succ_{\mathsf{mul}} t'_{i+1}$ for all $i \geqslant 1$. However, this is excluded because $\succ$ is well-founded on $\{0, \ldots, c\}$ and hence $\succ_{\mathsf{mul}}$ is well-founded on $\mathcal{T}(\mathcal{F}_{\{0,\ldots,c\}}, \mathcal{V})$ [3]. To prove the second part of the theorem consider an arbitrary (finite) rewrite sequence

$$t \rightarrow_{\mathcal{R}/\mathcal{S}} t_1 \rightarrow_{\mathcal{R}/\mathcal{S}} \cdots \rightarrow_{\mathcal{R}/\mathcal{S}} t_m$$

with $t \in L$. Similar as before this sequence can be lifted to a $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ sequence

$$t' \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})} t'_1 \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})} \cdots \rightarrow_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})} t'_m$$

of the same length starting from $t' = \mathsf{lift}_0(t)$ such that $t'_i \succ_{\mathsf{mul}} t'_{i+1}$ for all $i \geqslant 0$. Here $t'_0 = t'$. Because $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for L, we know that there is a $c \geqslant 0$ such that the height of every function symbol occurring in a term in the lifted sequence is at most c. Let k be the maximal number of function symbols occurring in some right-hand side in $\mathcal{R} \cup \mathcal{S}$. Due to a remark in [4] we know that the length of the $\succ_{\mathsf{mul}}$ chain from t' to t'_m is bounded by $\|t'\| \cdot (k+1)^c$. Since $\|t'\| = \|t\|$ and both, the lifted as well as the original sequence are as long as the $\succ_{\mathsf{mul}}$ chain starting at t', we conclude that the length of the $\mathcal{R}/\mathcal{S}$ rewrite sequence starting at the term t is bounded by $\|t\| \cdot (k+1)^c$. ∎

Corollary 4.10. *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS such that $\mathcal{R}$ and $\mathcal{S}$ are linear and let L be a set of terms. If $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for L then $(\to_{\mathcal{R}/\mathcal{S}}, \to_{\mathcal{S}})$ is a complexity pair. Furthermore $\mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}}, L)$ is bounded by a linear polynomial for any term $t \in L$.*

Proof. Follows immediately from Theorem 4.9. ∎

To prove that a relative TRS $\mathcal{R}/\mathcal{S}$ is match-RT-bounded for a set of terms L we construct a tree automaton $\mathcal{A}$ that is compatible with the rewrite rules of $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$ and $\mathsf{lift}_0(L)$. Since the set $\to^*_{\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})}(\mathsf{lift}_0(L))$ need not be regular, even for left-linear $\mathcal{R}$ and $\mathcal{S}$ and regular L (see [9]) we cannot hope to give an exact automaton construction. The general idea [8, 9] is to look for violations of the compatibility requirement: $l\sigma \to^*_\Delta q$ and $r\sigma \not\to^*_\Delta q$ for some rewrite rule $l \to r$, state substitution σ, and state q. Then we add new states and transitions to the current automaton to ensure $r\sigma \to^*_\Delta q$. After $r\sigma \to^*_\Delta q$ has been established, we repeat this process until a compatible automaton is obtained. Note that this may never happen if new states are repeatedly added.

Example 4.11. We show that the relative TRS $\mathcal{R}/\mathcal{S}$ of Example 4.6 is match-RT-bounded by constructing a compatible tree automaton. As starting point we consider the automaton

$$\mathsf{nil}_0 \to 1 \qquad \mathsf{rev}_0(1) \to 1 \qquad \mathsf{cons}_0(1,1) \to 1 \qquad \mathsf{rev}'_0(1,1) \to 1$$

which accepts the set of all ground terms over the signature $\{\mathsf{nil}, \mathsf{rev}, \mathsf{cons}, \mathsf{rev}'\}$. Since $\mathsf{rev}_0(x) \to_{\mathsf{match}(\mathcal{R})} \mathsf{rev}'_1(x, \mathsf{nil}_1)$ and $\mathsf{rev}_0(1) \to 1$, we add the transitions $\mathsf{nil}_1 \to 2$ and $\mathsf{rev}'_1(1,2) \to 1$. At next consider the rule $\mathsf{rev}'_1(\mathsf{nil}_0, y) \to_{\mathsf{match}(\mathcal{R})} y$ with $\mathsf{rev}'_1(\mathsf{nil}_0, 2) \to^* 1$. In order to solve this compatibility violation we add the transition $2 \to 1$. After that we consider the compatibility violation $\mathsf{rev}'_1(\mathsf{cons}_0(1,1), 2) \to^* 1$ but $\mathsf{rev}'_1(1, \mathsf{cons}_1(1,2)) \not\to^* 1$ caused by the rule $\mathsf{rev}'_1(\mathsf{cons}_0(x,y), z) \to_{\mathsf{match\text{-}RT}(\mathcal{S})} \mathsf{rev}'_1(y, \mathsf{cons}_1(x,z))$. In order to ensure $\mathsf{rev}'_1(1, \mathsf{cons}_1(1,2)) \to^* 1$ we reuse the transition $\mathsf{rev}'_1(1,2) \to 1$ and add the new transition $\mathsf{cons}_1(1,2) \to 2$. Finally $\mathsf{rev}'_0(\mathsf{cons}_1(x,y), z) \to_{\mathsf{match\text{-}RT}(\mathcal{S})} \mathsf{rev}'_1(y, \mathsf{cons}_1(x,z))$ and the derivation $\mathsf{rev}'_0(\mathsf{cons}_1(1,2), 1) \to^* 1$ give rise to the transition $\mathsf{cons}_1(1,1) \to 2$. After this step, the obtained tree automaton is compatible with $\mathsf{match\text{-}RT}(\mathcal{R}/\mathcal{S})$. Hence $\mathcal{R}/\mathcal{S}$ is match-RT-bounded by 1. Due to Corollary 4.10 we can conclude linear derivational complexity of $\mathcal{R}/\mathcal{S}$. We remark that the ordinary match-bound technique fails on $\mathcal{R}/\mathcal{S}$ because $\mathcal{R} \cup \mathcal{S}$ induces quadratic derivational complexity (consider the term $\mathsf{rev}^n(x)\sigma^m$ for $\sigma = \{x \mapsto \mathsf{cons}(y,x)\}$).

4.2. Beyond Complexity Pairs

An obvious question is whether it suffices to estimate the complexity of $\mathcal{R}_1/\mathcal{S}_1$ and $\mathcal{R}_2/\mathcal{S}$ (in contrast to $\mathcal{R}_2/\mathcal{S}_2$) to conclude some complexity bound for $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$. The following example by Hofbauer [15] shows that in general the complexity of $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$

might be much larger than the sum of the components; even for systems where both parts have linear derivational complexity.

Example 4.12. Consider the TRS $\mathcal{R}_1$ consisting of the single rule $\mathsf{c}(\mathsf{L}(x)) \to \mathsf{R}(x)$ and the TRS $\mathcal{R}_2$ consisting of the rules

$$\mathsf{R}(\mathsf{a}(x)) \to \mathsf{b}(\mathsf{b}(\mathsf{R}(x))) \qquad \mathsf{R}(x) \to \mathsf{L}(x) \qquad \mathsf{b}(\mathsf{L}(x)) \to \mathsf{L}(\mathsf{a}(x))$$

Here $\mathcal{S} = \varnothing$. The derivational complexity of the relative TRS $\mathcal{R}_1/\mathcal{S}_1$ is linear, due to the SLI that just counts the c's. The derivational complexity of $\mathcal{R}_2/\mathcal{S}$ is linear as well since the system is match-bounded by 2. However, the relative TRS $\mathcal{R}/\mathcal{S}$ with $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ admits exponentially long $\mathcal{R}$-derivations in the size of the starting term:

$$\begin{aligned}\mathsf{c}^n(\mathsf{L}(\mathsf{a}(x))) &\to \mathsf{c}^{n-1}(\mathsf{R}(\mathsf{a}(x))) \to \mathsf{c}^{n-1}(\mathsf{b}(\mathsf{b}(\mathsf{R}(x)))) \to \mathsf{c}^{n-1}(\mathsf{b}(\mathsf{b}(\mathsf{L}(x))))\\ &\to \mathsf{c}^{n-1}(\mathsf{b}(\mathsf{L}(\mathsf{a}(x)))) \to \mathsf{c}^{n-1}(\mathsf{L}(\mathsf{a}(\mathsf{a}(x)))) \to^* \mathsf{L}(\mathsf{a}^{2^n}(x))\end{aligned}$$

Under certain circumstances the problem from the preceding example does not occur. Although developed for a different setting (to estimate weak dependency pair steps relative to usable rule steps) the weight gap principle of Hirokawa and Moser [12] gives a sound criterion when estimating complexity bounds as shown above. Among non-duplication of $\mathcal{R}_1$ the theorem requires a *special* linear bound on $\mathcal{R}_2$. Then the derivational complexity of $\mathcal{R}_1/\mathcal{R}_2$ determines the derivational complexity of $\mathcal{R}_1 \cup \mathcal{R}_2$. However, in contrast to runtime complexity, for derivational complexity non-duplication is no (real) restriction since any duplicating TRSs immediately admits exponentially long derivations. Because we focus on feasible upper bounds all systems considered are non-duplicating.

Theorem 4.13 (Hirokawa and Moser [12])**.** *Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be TRSs, $\mathcal{R}_1$ be non-duplicating, and let $\mathcal{M}$ be an SLI such that $\mathcal{R}_2 \subseteq \succ_{\mathcal{M}}$. Then for all $\mathcal{R}_1$ and $\mathcal{M}$ there exist constants K and L such that*

$$K \cdot \mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{R}_2}) + L \cdot |t| \geqslant \mathsf{dl}(t, \to_{\mathcal{R}_1 \cup \mathcal{R}_2})$$

whenever t is terminating on $\mathcal{R}_1 \cup \mathcal{R}_2$. ∎

The next corollary generalizes this result to the relative rewriting setting.

Corollary 4.14. *Let $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$ be a relative TRS, $\mathcal{R}_1$ be non-duplicating, and let $\mathcal{M}$ be an SLI such that $\mathcal{R}_2 \subseteq \succ_{\mathcal{M}}$ and $\mathcal{S} \subseteq \succeq_{\mathcal{M}}$. Then for all $\mathcal{R}_1$ and $\mathcal{M}$ there exist constants K and L such that*

$$K \cdot \mathsf{dl}(t, \to_{\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S})}) + L \cdot |t| \geqslant \mathsf{dl}(t, \to_{(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}})$$

whenever t is terminating on $(\mathcal{R}_1 \cup \mathcal{R}_2)/\mathcal{S}$.

Proof. Follows from Theorem 4.13. ∎

An immediate consequence of the above corollary is that for any relative TRS $\mathcal{R}/\mathcal{S}$ we can shift rewrite rules in $\mathcal{R}$ that are strictly oriented by an SLI $\mathcal{M}$ into the $\mathcal{S}$ component, provided that $\mathcal{R}$ is non-duplicating and all rules in $\mathcal{S}$ behave nicely with respect to $\succeq_{\mathcal{M}}$. Note that the above corollary does not require that all rules from $\mathcal{R}$ are (strictly) oriented. This causes some kind of non-determinism which is demonstrated in the next example.

Example 4.15. Consider the TRSs $\mathcal{R} = \{\mathsf{a}(\mathsf{b}(x)) \to \mathsf{a}(\mathsf{c}(x)), \mathsf{d}(\mathsf{c}(x)) \to \mathsf{d}(\mathsf{b}(x))\}$ and $\mathcal{S} = \varnothing$. Obviously there are two SLIs that allow to preprocess $\mathcal{R}/\mathcal{S}$. Counting b's results in the relative TRS $\mathcal{R}_2/\mathcal{S}_2$ whereas counting c's yields $\mathcal{R}_1/\mathcal{S}_1$. Note that both results are different and cannot be further processed by Corollary 4.14.

5. Assessment

In the first part of this section we prove that the modular setting based on Theorem 4.4 is strictly more powerful in theory than the direct approach. (For gains in power in practice, cf. Section 7.) To make the reasoning easier we assume full (in contrast to relative) rewriting and that Theorem 4.4 has been applied iteratively, until every component $\mathcal{R}_i$ consists of a single rule. The next lemma states that if only TMIs of dimension one are employed, then in theory there is no difference in power between the two settings.

Lemma 5.1. *Let $\mathcal{R}$ be a TRS. There exists a TMI $\mathcal{M}$ of dimension one compatible with $\mathcal{R}$ if and only if there exists a family of TMIs $\mathcal{M}_i$ of dimension one such that $\mathcal{M}_i$ is compatible with $\mathcal{R}_i/\mathcal{S}_i$ for any i.*

Proof. The implication from left to right obviously holds since $\mathcal{M}$ is a suitable candidate for every $\mathcal{M}_i$. For the reverse direction we construct a TMI $\mathcal{M}$ compatible with $\mathcal{R}$ based on the family of TMIs $\mathcal{M}_i$. It is straightforward to check that defining $f_{\mathcal{M}}(x_1, \ldots, x_m) = \sum_i f_{\mathcal{M}_i}(x_1, \ldots, x_m)$ for any $f \in \mathcal{F}$ yields a TMI of dimension one compatible with $\mathcal{R}$. ∎

Due to Theorem 4.4 the complexity is not affected when using the modular setting. Hence when using TMIs of dimension one in theory both approaches can prove the same bounds. But experiments in Section 7 show that in practice proofs are easier to find in the modular setting since, e.g., the coefficients of the interpretations can be chosen smaller. For larger dimensions just the only-if direction of Lemma 5.1 holds. To see this we consider the TRS Strategy_removed_AG01/#4.21.[2] For this system a modular complexity proof based on TMIs of dimension two (see web site in Footnote 5 on page 398) yields a quadratic upper bound on the derivational complexity. However, due to the next lemma no such proof is possible in the direct setting. (We remark that there exist TMIs of dimension three that are compatible with this TRS).

Lemma 5.2. *The TRS* Strategy_removed_AG01/#4.21 *does not admit a TMI of dimension two compatible with it.*

Proof. To address all possible interpretations we extracted the set of constraints that represent a TMI of dimension two compatible with the TRS. MiniSmt[3] can detect unsatisfiability of these constraints. Details can be found at the web site in Footnote 5 on page 398. ∎

The next result shows that any direct proof transfers into the modular setting without increasing the bounds on the derivational complexity.

Lemma 5.3. *Let $\succ$ be a finitely branching rewrite relation and let $\mathcal{R}$ be a TRS compatible with $\succ$. Then for any i there exits a complexity pair $(\succ_i, \succsim_i)$ which is compatible with the relative TRS $\mathcal{R}_i/\mathcal{S}_i$. Furthermore for any term t we have $\mathcal{O}(\mathsf{dl}(t, \succ)) = \mathsf{max}_i\{\mathcal{O}(\mathsf{dl}(t, \succ_i))\}$.*

Proof. Fix i. Let $(\succ_i, \succsim_i)$ be $(\succ, =)$. It is easy to see that $(\succ, =)$ is a complexity pair because $\succ$ and $=$ are compatible rewrite relations. It remains to show that $\mathcal{O}(\mathsf{dl}(t, \succ)) = \mathsf{max}_i\{\mathcal{O}(\mathsf{dl}(t, \succ_i))\}$. To this end we observe that $\sum_i(\mathsf{dl}(t, (\succ_i, \succsim_i)) = n \cdot \mathsf{dl}(t, \succ)$ for all terms t. Here n denotes the number of complexity pairs. Basic properties of the $\mathcal{O}$-notation yield the desired result. ∎

[2] Labels in sans-serif font refer to TRSs from the TPDB 7.0.2, see http://termination-portal.org.

[3] http://cl-informatik.uibk.ac.at/software/minismt

As a corollary we obtain that in general the modular complexity setting is more powerful than the direct one. Even when applying just TMIs of the same dimension modularly.

Corollary 5.4. *The modular complexity setting is strictly more powerful than the direct one.*

Proof. Due to Lemmata 5.2 and 5.3. and the discussion before Lemma 5.2. ∎

As already mentioned earlier, the modular setting does not only allow to combine TMIs (of different dimensions) in one proof but even different complexity criteria. This is illustrated in the next example.

Example 5.5. Consider the TRS $\mathcal{R}$ (Transformed_CSR_04/Ex16_Luc06_GM) with the rules:

$$\mathsf{c} \to \mathsf{a} \qquad \mathsf{mark}(\mathsf{a}) \to \mathsf{a} \qquad \mathsf{g}(x,y) \to \mathsf{f}(x,y)$$
$$\mathsf{c} \to \mathsf{b} \qquad \mathsf{mark}(\mathsf{b}) \to \mathsf{c} \qquad \mathsf{g}(x,x) \to \mathsf{g}(\mathsf{a},\mathsf{b}) \qquad \mathsf{mark}(\mathsf{f}(x,y)) \to \mathsf{g}(\mathsf{mark}(x),y)$$

By using Corollary 4.14 twice, we can transform $\mathcal{R}$ into the relative TRS $\mathcal{R}'/\mathcal{S}'$ where $\mathcal{R}' = \mathcal{R}_1 \cup \mathcal{R}_2$, $\mathcal{S}' = \mathcal{R} \setminus \mathcal{R}'$, $\mathcal{R}_1 = \{\mathsf{g}(x,x) \to \mathsf{g}(\mathsf{a},\mathsf{b})\}$, and $\mathcal{R}_2 = \{\mathsf{mark}(\mathsf{f}(x,y)) \to \mathsf{g}(\mathsf{mark}(x),y)\}$. After that we can apply match-RT-bounds (extended to non-duplicating TRSs) to show that the derivational complexity induced by the relative TRS $\mathcal{R}_1/\mathcal{S}_1$ is at most linear. Finally, by applying Theorem 4.4 with TMIs of dimension two to the relative TRS $\mathcal{R}_2/\mathcal{S}_2$ we obtain a quadratic upper bound on the derivational complexity of $\mathcal{R}$. Further details of the proof can be accessed from the web site accompanying this paper (see Footnote 5 on page 398). The quadratic bound is tight as $\mathcal{R}$ admits derivations

$$\mathsf{mark}^n(x)\sigma^m \to^m \mathsf{mark}^{n-1}(x)\tau^m\gamma \to^m \mathsf{mark}^{n-1}(x)\sigma^m\gamma \to^{2m(n-1)} x\sigma^m\gamma^n$$

of length $2mn$ where $\sigma = \{x \mapsto \mathsf{f}(x,y)\}$, $\tau = \{x \mapsto \mathsf{g}(x,y)\}$, and $\gamma = \{x \mapsto \mathsf{mark}(x)\}$. Last but not least we remark that none of the involved techniques can establish an upper bound on its own. In case of match-bounds this follows from the fact that $\mathcal{R}$ admits quadratic derivational complexity. The same reason also holds for Corollary 4.14 because SLIs induce linear complexity bounds. Finally, TMIs fail because they cannot orient the rewrite rule $\mathsf{g}(x,x) \to \mathsf{g}(\mathsf{a},\mathsf{b})$.

Next we consider the TRS Zantema_04/z086. The question about the derivational complexity of it has been stated as problem #105 on the RTA LooP.[4]

Example 5.6. Consider the TRS $\mathcal{R}$ (Zantema_04/z086) consisting of the rules:

$$\mathsf{a}(\mathsf{a}(x)) \to \mathsf{c}(\mathsf{b}(x)) \qquad \mathsf{b}(\mathsf{b}(x)) \to \mathsf{c}(\mathsf{a}(x)) \qquad \mathsf{c}(\mathsf{c}(x)) \to \mathsf{b}(\mathsf{a}(x))$$

Adian showed in [1] that $\mathcal{R}$ admits at most quadratic derivational complexity. Since the proof is based on a low-level reasoning on the structure of $\mathcal{R}$, it is specific to this TRS and challenging for automation. With our approach we cannot prove the quadratic bound on the derivational complexity of $\mathcal{R}$. However, Corollary 4.14 permits to establish some progress. Using an SLI counting a's and b's it suffices to determine the derivational complexity of $\mathcal{R}_3/\mathcal{S}_3$. This means that it suffices to bound how often the rule $\mathsf{c}(\mathsf{c}(x)) \to \mathsf{b}(\mathsf{a}(x))$ is applied relative to the other rules. The benefit is that now, e.g., a TMI must only orient one rule strictly and the other two rules weakly (compared to all three rules strictly). It has to be clarified if the relative formulation of the problem can be used to simplify the proof in [1].

The next example shows that although the modular approach permits lower bounds compared to the old one further criteria for splitting TRSs should be investigated.

[4] `http://rtaloop.mancoosi.univ-paris-diderot.fr`

Example 5.7. Consider the TRS $\mathcal{R}$ (SK90/4.30) consisting of the following rules:

$$\begin{array}{lll} \mathsf{f}(\mathsf{nil}) \to \mathsf{nil} & \mathsf{f}(\mathsf{nil} \circ y) \to \mathsf{nil} \circ \mathsf{f}(y) & \mathsf{f}((x \circ y) \circ z) \to \mathsf{f}(x \circ (y \circ z)) \\ \mathsf{g}(\mathsf{nil}) \to \mathsf{nil} & \mathsf{g}(x \circ \mathsf{nil}) \to \mathsf{g}(x) \circ \mathsf{nil} & \mathsf{g}(x \circ (y \circ z)) \to \mathsf{g}((x \circ y) \circ z) \end{array}$$

In [19] a TMI compatible with $\mathcal{R}$ of dimension four is given showing that the derivational complexity is bounded by a polynomial of degree four. Using Theorem 4.4 with TMIs of dimension three yields a cubic upper bound (see web site in Footnote 5 on page 398). Although our approach enables showing a lower complexity than [19] this bound is still not tight since the derivational complexity of $\mathcal{R}$ is quadratic. For this particular TRS it is easy to observe that rules defining f do not interfere with rules defining g and vice versa. Thereby we could estimate the derivational complexity of $\mathcal{R}$ by bounding the two TRSs defining f and g separately.

6. Implementation

To estimate the derivational complexity of a TRS $\mathcal{R}$ we first transform $\mathcal{R}$ into the relative TRS $\mathcal{R}/\varnothing$. If the input is already a relative TRS this step is omitted. Afterwards we try to estimate the derivational complexity of $\mathcal{R}/\mathcal{S}$ by splitting $\mathcal{R}$ into TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $\mathsf{dl}(t, \to_{\mathcal{R}_1/\mathcal{S}_1}) + \mathsf{dl}(t, \to_{\mathcal{R}_2/\mathcal{S}_2}) \geqslant \mathsf{dl}(t, \to_{\mathcal{R}/\mathcal{S}})$. This is done by moving step by step those rewrite rules from $\mathcal{R}$ to $\mathcal{S}$ of which the derivational complexity can be bounded. In each step a different technique can be applied. As soon as $\mathcal{R}$ is empty, we can compute the derivational complexity of $\mathcal{R}/\mathcal{S}$ by summing up all intermediate results. In the following we describe the presented approach more formally and refer to it as the *complexity framework*.

A *complexity problem* (CP problem for short) is a pair $(\mathcal{R}/\mathcal{S}, f)$ consisting of a relative TRS $\mathcal{R}/\mathcal{S}$ and a unary function f. In order to prove the derivational complexity of a CP problem so-called *CP processors* are applied. A CP processor is a function that takes a CP problem $(\mathcal{R}/\mathcal{S}, f)$ as input and returns a new CP problem $(\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S}), f')$ as output. Here $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. In order to be employed to prove an upper bound on the complexity of a given CP problem they need to be *sound*: $f(n) + \mathsf{dc}(n, \to_{\mathcal{R}/\mathcal{S}}) \in \mathcal{O}(f'(n) + \mathsf{dc}(n, \to_{\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S})}))$. To ensure that a CP processor can be used to prove a lower-bound on the derivational complexity of a given CP problem it must be *complete*, i.e., $f(n) + \mathsf{dc}(n, \to_{\mathcal{R}/\mathcal{S}}) \in \Omega(f'(n) + \mathsf{dc}(n, \to_{\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S})}))$. From the preceding sections the following CP processors can be derived.

Theorem 6.1. *The CP processor*

$$(\mathcal{R}/\mathcal{S}, f) \mapsto \begin{cases} (\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S}), f') & \textit{if } \mathcal{R}_2 \subseteq \succ \textit{ and } \mathcal{R}_1 \cup \mathcal{S} \subseteq \succeq \\ & \textit{for some complexity pair } (\succ, \succeq) \\ (\mathcal{R}/\mathcal{S}, f) & \textit{otherwise} \end{cases}$$

where $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ *and* $f'(n) = f(n) + \mathsf{dc}(n, \succ)$ *is sound.*

Proof. Follows from Corollary 3.3 and Theorem 4.4. ∎

Table 1: Derivational complexity of 1172 TRSs

	p	l	q	c	time	timeout
direct	287	180	87	20	0.72	192
modular	311	198	86	27	1.19	340

Note that the above theorem defines a general version of a CP processor based on a complexity pair $(\succ, \succsim)$. To obtain concrete instances of this CP processor one can use for example the complexity pairs from Theorem 3.4 and Corollary 4.10.

Theorem 6.2. *The CP processor*

$$(\mathcal{R}/\mathcal{S}, f) \mapsto \begin{cases} (\mathcal{R}_1/(\mathcal{R}_2 \cup \mathcal{S}), f') & \text{if } \mathcal{R}_1 \text{ is non-duplicating and} \\ & \mathcal{R}_2 \subseteq {\succ_{\mathcal{M}}} \text{ and } \mathcal{S} \subseteq {\succsim_{\mathcal{M}}} \text{ for some SLI } \mathcal{M} \\ (\mathcal{R}/\mathcal{S}, f) & \text{otherwise} \end{cases}$$

where $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ *and* $f'(n) = f(n) + n$ *is sound.*

Proof. Straightforward by Corollary 4.14. ∎

7. Experimental Results

The techniques described in the preceding sections are implemented in the complexity analyzer CaT (freely available from `http://cl-informatik.uibk.ac.at/software/cat`) which is built on top of $\mathsf{T_TT_2}$ [18], a powerful termination tool for TRSs. Both tools are written in OCaml and interface MiniSat [6] and Yices [5].

Below we report on the experiments we performed with CaT on the 1172 TRSs in version 7.0.2 of the TPDB that are non-duplicating.[5] All tests have been performed on a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and on 64 GB of main memory. We remark that similar results have been obtained on a dual-core laptop. Our results are summarized in Table 1. Here, direct refers to the conventional setting where all rules must be oriented at once whereas modular first transforms a TRS $\mathcal{R}$ into a relative TRS $\mathcal{R}/\varnothing$ before the CP processors from Section 6 are employed. As base methods we use the match-bounds technique as well as TMIs and arctic matrix interpretations [16] of dimensions one to three. The latter two are implemented by bit-blasting arithmetic operations to SAT [7, 16]. The coefficients of a matrix of dimension d are represented in $6 - d$ bits, one additional bit is used for intermediate results. All base methods are run in parallel, but criteria that yield larger derivational complexity are executed slightly delayed. This allows to maximize the number of low bounds.

Table 1 shows that the modular approach allows to prove significantly more polynomial bounds (column p) for the systems; furthermore these bounds are also smaller. Here column l refers to linear, q to quadratic, and c to cubic derivational complexity. We also list the average time (in seconds) needed for finding a bound and the number of timeouts, i.e, when the tool did not deliver an answer within 60 seconds. The modular setting is slower since there typically more proofs are required to succeed. However, we anticipate that by making use of incremental SAT solvers (for the matrix methods) the time can be reduced.

[5] Full details available from `http://cl-informatik.uibk.ac.at/software/cat/modular`.

Table 2: Termination competition 2009 (derivational complexity)

	points	p	l	q	c	r
CaT	540	137	84	41	7	5
Matchbox	397	102	44	52	5	1
TCT	380	109	32	69	8	0

For a comparison of our method with other tools we refer the reader to the international termination competition (referenced in Footnote 1 on page 385). In 2008 and 2009, CaT won all categories related to derivational complexity analysis. Note that in 2009 it used a slightly weaker implementation of Theorem 4.4 than presented here and did not implement Corollary 4.14; especially the latter speeds up proofs. Some statistics from the 2009 competition are listed in Table 2. The total points are computed by a scoring scheme that favors tools that yield small upper bounds. The column r indicates polynomial bounds of degree four.

8. Conclusion

In this paper we have introduced a modular approach for estimating the derivational complexity of TRSs by considering relative rewriting. We showed how existing criteria (for full rewriting) can be lifted into the relative setting without much effort. The modular approach is easy to implement and has been proved strictly more powerful than traditional methods in theory and practice. Since the modular method allows to combine different criteria, typically smaller complexity bounds are achieved. All our results directly extend to more restrictive notions of complexity, e.g., runtime complexity (see below). We stress that as a by-product of our investigations we obtained an alternative approach to [23] that can be used to prove relative termination using match-bounds. It remains open which of the two approaches is more powerful. Finally we remark that our setting allows a more fine-grained complexity analysis, i.e., while traditionally a quadratic derivational complexity ensures that any rule is applied at most quadratically often, our approach can make different statements about single rules. Hence even if a proof attempt does not succeed completely, it may highlight the problematic rules.

As related work we mention [15] which also considers relative rewriting for complexity analysis. However, there the complexity of $\mathcal{R}_1 \cup \mathcal{R}_2$ is investigated by considering $\mathcal{R}_1/\mathcal{R}_2$ and $\mathcal{R}_2$. Hence [15] also gives rise to a modular reasoning but the obtained complexities are typically beyond polynomials. For runtime complexity analysis Hirokawa and Moser [12,13] consider weak dependency pair steps relative to the usable rules, i.e., $\mathsf{WDP}(\mathcal{R})/\mathsf{UR}(\mathcal{R})$. However, since in the current formulation of weak dependency pairs some complexity might be hidden in the usable rules they do not really obtain a relative problem. As a consequence they can only apply restricted criteria for the usable rules. Note that our approach can directly be used to show bounds on $\mathsf{WDP}(\mathcal{R})/\mathsf{UR}(\mathcal{R})$ by considering $\mathsf{WDP}(\mathcal{R}) \cup \mathsf{UR}(\mathcal{R})$. Due to Corollary 4.14 this problem can be transformed into an (unrestricted) relative problem $\mathsf{WDP}(\mathcal{R})/\mathsf{UR}(\mathcal{R})$ whenever the constraints in [12] are satisfied. Moreover, if somehow the *problematic* usable rules could be determined and shifted into the $\mathsf{WDP}(\mathcal{R})$ component, then this *improved* version of weak dependency pairs corresponds to a relative problem without additional restrictions, admitting further benefit from our contributions.

We plan to investigate if the arctic matrix method also satisfies the weight gap principle. Other promising techniques that (might) allow a modular processing of subsystems comprise criteria for constructor sharing TRSs [20]. Last but not least it seems feasible to combine the approach for relative match-bounds introduced in [23] with the one presented in Section 4.1. By doing so, a stronger version of relative match-bounds could be obtained.

Acknowledgments. We thank Johannes Waldmann for communicating Example 4.12.

References

[1] Adian, S.I.: Upper bound on the derivational complexity in some word rewriting system. Doklady Math. 80(2), 679–683 (2009)

[2] Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)

[3] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)

[4] Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Comm. ACM 22(8), 465–476 (1979)

[5] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: CAV 2006. LNCS, vol. 4144, pp. 81–94 (2006)

[6] Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT 2004. LNCS, vol. 2919, pp. 502–518 (2004)

[7] Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. JAR 40(2-3), 195–220 (2008)

[8] Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: RTA 1998. LNCS, vol. 1379, pp. 151–165 (1998)

[9] Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. I&C 205(4), 512–534 (2007)

[10] Geser, A.: Relative termination. PhD thesis, Universität Passau, Germany (1990). Available as: Report 91-03, Ulmer Informatik-Berichte, Universität Ulm, 1991

[11] Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C 199(1-2), 172–199 (2005)

[12] Hirokawa, N., Moser, G.: Automated complexity analysis based on the dependency pair method. In: IJCAR 2008. LNCS, vol. 5195, pp. 364–379 (2008)

[13] Hirokawa, N., Moser, G.: Complexity, graphs, and the dependency pair method. In: LPAR 2008. LNCS (LNAI), vol. 5330, pp. 652–666 (2008)

[14] Hofbauer, D., Lautemann, C.: Termination proofs and the length of derivations (preliminary version). In: RTA 1989. LNCS, vol. 355, pp. 167–177 (1989)

[15] Hofbauer, D., Waldmann, J.: Complexity bounds from relative termination proofs. Talk at the Workshop on Proof Theory and Rewriting, Obergurgl (2006). Available from `http://www.imn.htwk-leipzig.de/~waldmann/talk/06/rpt/rel/main.pdf`

[16] Koprowski, A., Waldmann, J.: Max/plus tree automata for termination of term rewriting. AC 19(2), 357–392 (2009)

[17] Korp, M., Middeldorp, A.: Match-bounds revisited. I&C 207(11), 1259–1283 (2009)

[18] Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: RTA 2009. LNCS, vol. 5595, pp. 295–304 (2009)

[19] Moser, G., Schnabl, A., Waldmann, J.: Complexity analysis of term rewriting based on matrix and context dependent interpretations. In: FSTTCS 2008. LIPIcs, vol. 2, pp. 304–315 (2008)

[20] Ohlebusch, E.: On the modularity of confluence of constructor-sharing term rewriting systems. In: CAAP 1994. LNCS, vol. 787, pp. 261–275 (1994)

[21] Terese: Term Rewriting Systems. vol. 55 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2003)

[22] Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen (2007). Available as technical report AIB-2007-17

[23] Waldmann, J.: Weighted automata for proving termination of string rewriting. J. Autom. Lang. Comb. 12(4), 545–570 (2007)

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 401-416
http://rewriting.loria.fr/rta/

PROVING PRODUCTIVITY IN INFINITE DATA STRUCTURES

HANS ZANTEMA [1,2] AND MATTHIAS RAFFELSIEPER [1]

[1] Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
E-mail address: `H.Zantema@tue.nl`
E-mail address: `M.Raffelsieper@tue.nl`

[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. For a general class of infinite data structures including streams, binary trees, and the combination of finite and infinite lists, we investigate the notion of productivity. This generalizes stream productivity. We develop a general technique to prove productivity based on proving context-sensitive termination, by which the power of present termination tools can be exploited. In order to treat cases where the approach does not apply directly, we develop transformations extending the power of the basic approach. We present a tool combining these ingredients that can prove productivity of a wide range of examples fully automatically.

1. Introduction

Some computations potentially go on forever. A standard example is the sieve of Eratosthenes producing the infinitely many prime numbers. The result of such a computation is then an infinite stream of elements. Although the computation itself goes on forever, there is a kind of termination involved that is called *productivity*: every finite initial part will be produced after a finite number of steps. We will consider computations specified by a number of rewrite rules that are interpreted as a lazy functional program. Then productivity can be characterized and investigated as a property of term rewriting, as was investigated before in [6, 11, 4, 19, 12].

Streams can be seen as infinite terms. Even when restricting to data structures representing the result of a computation, it is natural not to restrict to streams. In case the computation possibly ends, then the result is not a stream but a finite list, and when parallelism is considered, naturally infinite trees come in. In this paper we develop techniques for automatically proving productivity of specifications in a wide range of infinite data structures, including streams, the combination with finite lists, and several kinds of infinite trees. Earlier techniques specifically for stream specifications were given in [6, 4, 19]. A key idea of our approach is to prove productivity by proving termination of *context-sensitive rewriting* [17, 9], that is, the restricted kind of rewriting in which rewriting is disallowed inside particular arguments of particular symbols. As strong tools like AProVE [8] and μ-Term

© Hans Zantema and Matthias Raffelsieper
Creative Commons Non-Commercial No Derivatives License

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.401

[13] have been developed to prove termination of context-sensitive rewriting automatically, the power of these tools can now be exploited to prove productivity automatically. As the underlying technique is completely different from the technique of [6, 4], it is expected that both approaches have their own merits. Indeed, there are examples where the technique of [6, 4] fails and our technique succeeds. The comparison the other way around is hard to make as the technique of [6, 4] only applies for proving productivity for a single ground term while our technique applies for proving productivity for all ground terms.

Through this paper we consider two kinds of terms: finite terms and infinite terms. As the elements of the infinite data structures we intend to define are infinite terms, infinite terms are unavoidable here. On the other hand, terms occurring in our specifications are finite. Rewriting has been investigated both for finite and infinite terms. But rewriting finite terms is much easier and better understood than infinitary rewriting, and for many properties, like several variants of termination, there is strong tool support to investigate these properties. We follow the policy to use infinite terms only where necessary, and exploit understanding and tool support for rewriting finite terms as much as possible. In this way we need the concept of infinite terms, but not of infinitary rewriting. Following this policy, elements of infinite data structures over a data set $\mathcal{D}$ are considered as infinite terms in which elements of $\mathcal{D}$ act as constants, and the infinite terms are composed from constructor symbols taken from a set $\mathcal{C}$, and elements of $\mathcal{D}$. In this world of infinite terms we want to avoid that data elements are infinite terms themselves. For instance, in considering streams over natural numbers as infinite terms, we want to be able to consider a stream $0 : 1 : 2 : 3 : 4 : \cdots$, but we do not want the data elements in such a stream to be infinite terms like $\mathsf{s}^\infty(0)$. When specifying elements of infinite data structures over a data set $\mathcal{D}$, the set $\mathcal{D}$ may be described as the set of (finite) ground normal forms of some rewriting system R_d over data signature Σ_d. As an example, for natural numbers with $+$ we can choose $\Sigma_d = \{0, \mathsf{s}, +\}$, and R_d consists of the rules $0 + x \to x$ and $\mathsf{s}(x) + y \to \mathsf{s}(x + y)$. Apart from Σ_d and R_d the specification then is given by a set R_s of rewrite rules over $\mathcal{C} \cup \Sigma_d \cup \Sigma_s$, where Σ_s consists of constants and auxiliary operations to be introduced for the specification. For instance, for defining the above mentioned stream $\mathsf{nat} = 0 : 1 : 2 : 3 : 4 : \cdots$ we introduce an auxiliary function $\mathsf{f} \in \Sigma_s$ on streams that replaces each element by its successor, and specify nat by choosing R_s to consist of the two rules

$$\mathsf{nat} \to 0 : \mathsf{f}(\mathsf{nat}), \quad \mathsf{f}(x : \sigma) \to \mathsf{s}(x) : \mathsf{f}(\sigma).$$

Here we have $\mathcal{C} = \{:\}$, $\Sigma_d = \{0, \mathsf{s}\}$, $R_d = \emptyset$, $\Sigma_s = \{\mathsf{nat}, \mathsf{f}\}$, and x, σ are variable symbols of type data and stream, respectively. In this setting productivity means that for every n the initial term can be rewritten to a (finite) term in which all symbols on depth less than n are constructor symbols. This notion of productivity is consistent with stream productivity as in [6, 4, 19], formalizing the spirit of stream productivity as introduced in [15]. It is also consistent with productivity as defined in [11, 12] for a setting even more general than ours.

Proving productivity may be hard. For the sieve of Eratosthenes proving productivity is beyond the scope of fully automatic techniques as it depends on the fact that there are infinitely many prime numbers. Moreover, we can specify an extra stream by filtering out every element in this stream of prime numbers that is distinct from its predecessor plus 2. This yields a stream specification, easily expressed in the format of this paper, of which productivity is equivalent to the existence of infinitely many prime twins: a well-known open problem in number theory. As expected for such a format suitable for expressing a

well-known open problem, productivity is an undecidable property. This has been proved independently by several people in [5, 16].

In contrast to [6, 4], we focus on requiring productivity not only for a single initial term, like nat in the above example, but for all finite ground terms. An easy induction argument shows that productivity holds for all ground terms if and only if every ground term rewrites to a term of which the root is a constructor symbol. As in [19] this characterization is the basis of our productivity investigations, but now for more general infinite data structures than only streams. A main reason for the focus on productivity for all ground terms is this technical convenience. In many cases, however, productivity of a single ground terms is equivalent to productivity for all ground terms over the operations that are relevant for the single ground term. If this does not hold and one is interested in productivity for a single ground term, our approach fails while the approach of [6, 4] may succeed.

The paper is organized as follows. In Section 2 we introduce infinite terms and give examples of several infinite data structures consisting of infinite terms. In Section 3 we introduce our notions of proper specifications and productivity. Being interested in deterministic computations, in proper specifications we require the rewrite systems to be orthogonal. A first basic result (Theorem 3.4) states that a specification is productive if for all rules the root of the right-hand side is a constructor symbol. In Section 4 we relate productivity to context-sensitive rewriting. The main result (Theorem 4.1) states that a specification is productive if context-sensitive termination holds for the rules of the specification, where rewriting is only allowed in the data arguments of the constructor symbols, and in all arguments of the other symbols. For cases where these approaches fail, in Section 5 we investigate transformations that transform a proper specification into another one, such that productivity of the original specification can be concluded from productivity of the transformed specification, the latter typically proved by the basic techniques from the earlier sections. Through these sections we give several examples of specifications of streams and binary trees for which productivity is proved. For many of these, productivity cannot be proved by earlier techniques. In Section 6 we describe an implementation of our techniques, by which productivity of all productive examples presented in this paper can be proved fully automatically. We conclude in Section 7.

2. Infinite Terms

Intuitively, a term (both finite and infinite) is defined by saying which symbol is at which position. Here, a *position* $p \in \mathbf{N}^*$ is a finite sequence of natural numbers. In order to be a proper term, some requirements have to be satisfied as indicated in the following definition. As we will only consider infinite terms over a set $\mathcal{C}$ of constructors and a set $\mathcal{D}$ of data (disjoint from $\mathcal{C}$), our terms will be two-sorted[1]: a sort s for the (infinite) terms to be defined, and a sort d for the data. Every $f \in \mathcal{C}$ is assumed to be of type $d^n \times s^m \to s$ for some $n, m \in \mathbf{N}$. We write $\mathsf{ar}(d, f) = n$ and $\mathsf{ar}(s, f) = m$. We write $\perp$ for undefined.

Definition 2.1. A (possibly infinite) *term* of sort s over $\mathcal{C}, \mathcal{D}$ is defined to be a map $t : \mathbf{N}^* \to \mathcal{C} \cup \mathcal{D} \cup \{\perp\}$ such that

- the root $t(\epsilon)$ of the term t is a constructor symbol, so $t(\epsilon) \in \mathcal{C}$, and

[1] In [11, 12] an arbitrary many-sorted setting is proposed. Our approach easily generalizes to a more general many-sorted setting, but for notational convenience we restrict to the two-sorted setting.

- for all $p \in \mathbf{N}^*$ and all $i \in \mathbf{N}$ we have
$$t(pi) \in \mathcal{D} \iff t(p) \in \mathcal{C} \wedge 1 \leq i \leq \mathsf{ar}(d, t(p)), \text{ and}$$
$$t(pi) \in \mathcal{C} \iff t(p) \in \mathcal{C} \wedge \mathsf{ar}(d, t(p)) < i \leq \mathsf{ar}(d, t(p)) + \mathsf{ar}(s, t(p)).$$

So $t(pi) = \bot$ for all p, i not covered by the above two cases.

We write $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ for the set of all terms over $\mathcal{C}, \mathcal{D}$.

An alternative equivalent definition of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ can be given based on co-algebra. Another alternative uses metric completion, where infinite terms are limits of finite terms. However, for the results in this paper we do not need these alternatives.

A position $p \in \mathbf{N}^*$ satisfying $t(p) \in \mathcal{C}$ is called a *position of* t of sort s. A position $p \in \mathbf{N}^*$ satisfying $t(p) \in \mathcal{D}$ is called a *position of* t of sort d. The *depth* of a position $p \in \mathbf{N}^*$ is the length of p considered as a string.

The usual notion of finite term coincides with a term in this setting having finitely many positions, that is, $t(p) = \bot$ for all but finitely many p. In case $\mathsf{ar}(s, f) > 0$ for all $f \in \mathcal{C}$ then no finite terms exist. This holds for streams. In case $\mathsf{ar}(d, f) = 0$ for all $f \in \mathcal{C}$ then no position of sort $\mathcal{D}$ exist, and terms do not depend on $\mathcal{D}$.

For $f \in \mathcal{C}$ with $\mathsf{ar}(d, f) = n$, $\mathsf{ar}(s, f) = m$, n elements $u_1, \ldots, u_n \in \mathcal{D}$ and m terms $t_1, \ldots, t_m$ we write $f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for the term t defined by $t(\epsilon) = f$, $t(i) = u_i$ for every $i = 1, \ldots, n$, $t(ip) = t_{i-n}(p)$ for every $p \in \mathbf{N}^*$ and $i = n+1, \ldots, n+m$, and $t(ip) = \bot$ if $i \notin \{1, \ldots, n+m\}$, or $i \in \{1, \ldots, n\}$ and $p \neq \epsilon$.

Example 2.2 (Streams)**.** Let $\mathcal{D}$ be an arbitrary given non-empty data set, and let $\mathcal{C} = \{:\}$, with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$. Then $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ coincides with the usual notion of *streams* over $\mathcal{D}$, being functions from $\mathbf{N}$ to $\mathcal{D}$. More precisely, a function $f : \mathbf{N} \to \mathcal{D}$ gives rise to an infinite term t defined by $t(2^n) =\ :$ and $t(2^n 1) = f(n)$ for every $n \in \mathbf{N}$, and $t(w) = \bot$ for all other strings $w \in \mathbf{N}^*$. Conversely, every $t : \mathbf{N}^* \to \mathcal{C} \cup \mathcal{D}$ satisfying the requirements of the definition of a term is of this shape. Note that if $\#\mathcal{D} = 1$, then there exists only one such term.

In case $\mathcal{D}$ is finite, an alternative approach is not to consider the binary constructor ':', but unary constructors for every element of $\mathcal{D}$. In this approach $\mathcal{D}$ does not play a role and is irrelevant.

Example 2.3 (Finite and infinite lists)**.** Let $\mathcal{D}$ be an arbitrary given non-empty data set, and let $\mathcal{C} = \{:, \mathsf{nil}\}$, with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$ and $\mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$. Then $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ covers both the *streams* over $\mathcal{D}$ as in Example 2.2 and the usual (finite) lists. As in Example 2.2, a function $f : \mathbf{N} \to \mathcal{D}$ gives rise to an infinite term t defined by $t(2^n) =\ :$ and $t(2^n 1) = f(n)$ for every $n \in \mathbf{N}$, and $t(w) = \bot$ for all other strings $w \in \mathbf{N}^*$. The only way nil can occur is where $t(2^n) = \mathsf{nil}$ for some $n \geq 0$, $t(2^i) =\ :$ and $t(2^i 1) \in \mathcal{D}$ for every $i < n$, and $t(w) = \bot$ for all other strings $w \in \mathbf{N}^*$, in this way representing a finite list of length n. Conversely, every $t : \mathbf{N}^* \to \mathcal{C} \cup \mathcal{D}$ satisfying the requirements of the definition of a term is of one of these two shapes. In the literature this combination of finite and infinite lists is sometimes called *lazy lists*.

Example 2.4 (Binary trees)**.** For infinite binary trees several variants fit in our format. We will meet the following:

- Infinite binary trees with nodes labeled by $\mathcal{D}$ are obtained by choosing $\mathcal{C} = \{\mathsf{b}\}$ with $\mathsf{ar}(d, \mathsf{b}) = 1$ and $\mathsf{ar}(s, \mathsf{b}) = 2$. In Example 4.4 the nodes are labeled by $\mathcal{D} \times \mathcal{D}$, obtained by choosing $\mathsf{ar}(d, \mathsf{b}) = 2$ instead.

- The combination of finite and infinite binary trees with nodes labeled by $\mathcal{D}$ is obtained by choosing $\mathcal{C} = \{\mathsf{b}, \mathsf{nil}\}$ with $\mathsf{ar}(d, \mathsf{b}) = 1$, $\mathsf{ar}(s, \mathsf{b}) = 2$ and $\mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$. In Example 3.5 the nodes are unlabeled, obtained by choosing $\mathsf{ar}(d, \mathsf{b}) = 0$ instead.

3. Specifications and Productivity

Throughout this paper we use some basics of term rewriting as is introduced e.g. in [1, 17]. In particular, a term rewriting system (TRS) is called *orthogonal* if the left-hand sides of the rules do not overlap, and every variable occurs at most once in every left-hand side of a rule.

We consider *specifications* in order to define elements of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$. We do this for the special case where $\mathcal{D}$ consists of the ground normal forms of an orthogonal terminating TRS R_d over a signature Σ_d. Here all symbols of Σ_d are considered to be of sort $d^n \to d$ for some $n \geq 0$. For defining elements of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$ we introduce a set Σ_s of defined symbols of sort s, disjoint from $\mathcal{C}$, all being of sort $d^n \times s^m \to s$ for some $n, m \in \mathbf{N}$, just like the elements of $\mathcal{C}$. The real specification is given by a set R_s of rewrite rules of sort s being of a special shape. Although the goal is to define elements of $\mathsf{T}^\infty(\mathcal{C}, \mathcal{D})$, most times being infinite, all terms in the specification are finite, and rewriting always refers to rewriting finite terms. All terms are well-sorted, that is, for every symbol f occurring in a term the sort of the term on the i-th argument equals the sort expected at that argument.

Definition 3.1. A *proper specification* $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ consists of $\Sigma_d, \Sigma_s, \mathcal{C}, R_d$ as described above and a TRS R_s over $\Sigma_d \cup \mathcal{C} \cup \Sigma_s$ consisting of rules of the shape

$$f(u_1, \ldots, u_n, t_1, \ldots, t_m) \to t,$$

where

- $f \in \Sigma_s$ is of type $d^n \times s^m \to s$,
- for every $i = 1, \ldots, m$ the term t_i is either
  - a variable of sort s, or
  - $t_i = g(d_1, \ldots, d_k, \sigma_1, \ldots, \sigma_l)$ for some $g \in \mathcal{C}$ with $\mathsf{ar}(d, g) = k$ and $\mathsf{ar}(s, g) = l$, where $\sigma_1, \ldots, \sigma_l$ are variables of sort s, and $d_1, \ldots, d_k$ are terms over Σ_d,
- t is a (well-sorted) term of sort s,
- $R_s \cup R_d$ is orthogonal, and
- every term of the shape $f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for $f \in \Sigma_s$, $u_1, \ldots, u_n \in \mathcal{D}$, and in which every t_i is of the shape $g(u'_1, \ldots, u'_n, t'_1, \ldots, t'_m)$ for $g \in \mathcal{C}$ and $u'_1, \ldots, u'_n \in \mathcal{D}$, matches with the left-hand side of a rule from R_s.

Intuitively, the last bullet requires exhaustiveness of pattern matching, as is a standard requirement in functional programming. Orthogonality is required for forcing unicity of the result of computation. The second bullet requires simplicity of left-hand sides of rules; in case this restriction does not hold it can be obtained by unfolding the rules and introducing extra symbols.

A proper specification is therefore a generalization of proper stream specifications as given in [18, 19]. Fixing $\mathcal{C}, \mathcal{D}$, typically a proper specification will be given by R_d, R_s in which Σ_d, Σ_s and the arities are left implicit since they are implied by the terms occurring in R_d, R_s. If a proper specification is only given by R_s, then R_d is assumed to be empty. This is what we will do several times, starting in Example 3.5.

For a term $t = f(\cdots)$ we write $\mathsf{root}(t) = f$; the symbol f is called the *root* of t.

A specification is called *productive* for a given ground term of sort s if every finite part of the intended resulting infinite term can be computed in finitely many steps. As the intended resulting infinite term consists of constructor symbols and data elements, and all ground terms of sort d rewrite to data elements by assumption, this is equivalent to the following.

Definition 3.2. A proper specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ is *productive* for a ground term t of sort s if for every $k \in \mathbf{N}$ there is a reduction $t \to^*_{R_s \cup R_d} t'$ for which every symbol of sort s in t' on depth less than k is in $\mathcal{C}$.

An important consequence of productivity is *well-definedness*: the term admits a unique interpretation as an infinite term. Intuitively, existence follows from taking the limit of the process of computing a constructor on every level, and reduce data terms to normal form. Uniqueness follows form orthogonality. For an investigation of well-definedness of stream specifications we refer to [18].

As in [19], in this paper we are interested in productivity for all finite ground terms of sort s rather than a single one. The following proposition states that for this case reaching a constructor on every arbitrary depth is equivalent to reaching a constructor at the root. As the latter characterization is simpler, this is the basis of all further observations on productivity in this paper. In [11, 12] productivity is also required for infinite terms, being a stronger restriction than ours, see Example 4.2. Again we stress that in this section all terms are finite.

Proposition 3.3. *A specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ is productive for all ground terms of sort s if and only if every ground term t of sort s admits a reduction $t \to^*_{R_s \cup R_d} t'$ for which $\mathsf{root}(t') \in \mathcal{C}$.*

Proof. The "only if" direction of the proposition is obvious. For the "if" direction, we prove the following claim by induction on k.

> **Claim.** Let $k \in \mathbf{N}$, and for all ground terms t of sort s we have $t \to^*_{R_s \cup R_d} t'$ with $\mathsf{root}(t') \in \mathcal{C}$. Then $t \to^*_{R_s \cup R_d} t''$ for a term t'' in which every symbol of sort s on depth less than k is in $\mathcal{C}$.

If $k = 1$, then the claim directly holds by choosing $t'' = t'$.

Otherwise, we have $t \to^*_{R_s \cup R_d} t' = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ with $\mathsf{root}(t') = f \in \mathcal{C}$, with f of type $d^n \times s^m \to s$. Applying the induction hypothesis to $t_1, \ldots, t_m$ yields $t_i \to^*_{R_s \cup R_d} t''_i$ with all symbols of sort s in t''_i are on depth $< k - 1$, for $i = 1, \ldots, m$. Now

$$t \to^*_{R_s \cup R_d} f(u_1, \ldots, u_n, t_1, \ldots, t_m) \to^*_{R_s \cup R_d} f(u_1, \ldots, u_n, t''_1, \ldots, t''_m)$$

proves the claim. ∎

Our first theorem gives a simple syntactic criterion for productivity, which can also be seen as a particular case of the analysis of friendly nesting specifications as given in [4].

Theorem 3.4. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ be a proper specification in which for every $\ell \to r$ in R_s the term r is not a variable and $\mathsf{root}(r) \in \mathcal{C}$. Then $\mathcal{S}$ is productive.*

Proof. According to Proposition 3.3 for every ground term t of sort s it suffices to prove that $t \to^*_{R_s \cup R_d} t'$ for a term t' satisfying $\mathsf{root}(t') \in \mathcal{C}$. We do this by induction on t. Let $t = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for $m, n \geq 0$. If $f \in \mathcal{C}$ we are done. So we may assume $f \in \Sigma_s$.

As they are ground terms of sort d, all u_i rewrite to elements of $\mathcal{D}$. By the induction hypothesis, all t_i rewrite to terms with root in $\mathcal{C}$, and in which the arguments of sort d rewrite to elements of $\mathcal{D}$. Now by the last requirement of properness, the resulting term matches with the left-hand side of a rule from R_s. By the assumption, by rewriting according to this rule a term is obtained of which the root is in $\mathcal{C}$. ∎

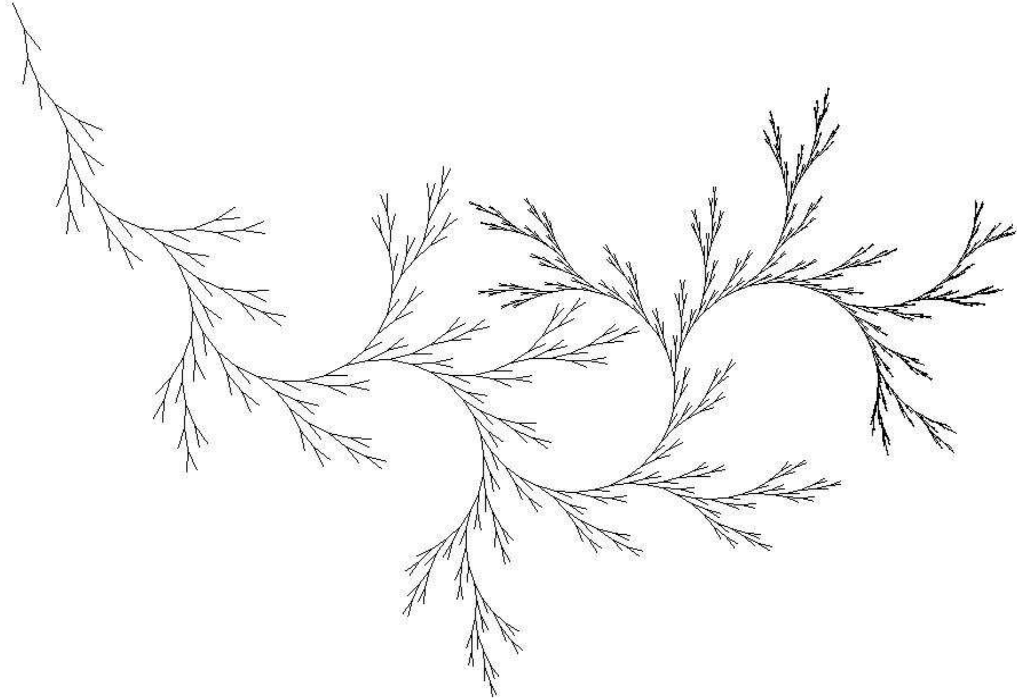
Example 3.5. Choose $\mathcal{C} = \{\mathsf{b}, \mathsf{nil}\}$ with $\mathsf{ar}(s, \mathsf{b}) = 2$ and $\mathsf{ar}(d, \mathsf{b}) = \mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$ representing the combination of finite and infinite unlabeled binary trees. Then

$$\mathsf{c} \to \mathsf{b}(\mathsf{b}(\mathsf{nil}, \mathsf{c}), \mathsf{c})$$

is a proper specification that is productive due to Theorem 3.4; the symbol c represents an infinite tree in which the number of nodes on depth n is exactly the n-th Fibonacci number. In the same setting

$$\begin{array}{rcl} \mathsf{p} & \to & \mathsf{b}(\mathsf{f}(\mathsf{p}), \mathsf{nil}) \\ \mathsf{f}(\mathsf{b}(x, y)) & \to & \mathsf{b}(\mathsf{f}(y), \mathsf{b}(\mathsf{nil}, \mathsf{f}(x))) \\ \mathsf{f}(\mathsf{nil}) & \to & \mathsf{nil} \end{array}$$

is a proper specification that is productive due to Theorem 3.4. The symbol p represents the infinite tree of which the initial part until depth 100 is shown in the following picture, in which the root of the tree is shown on top left:



4. Proving Productivity by Context-Sensitive Termination

As intended for generating infinite terms, the TRS $R_s \cup R_d$ will never be terminating. However, when disallowing rewriting inside arguments of sort s of constructor symbols, it may be terminating. The main result of this section states that if this is the case, then

the specification is productive. The variant of rewriting with the restriction that rewriting inside certain arguments of certain symbols is disallowed, is called *context-sensitive rewriting* [17, 9]. In context-sensitive rewriting, for every symbol f the set $\mu(f)$ of arguments of f is specified inside which rewriting is allowed. More precisely, μ-rewriting $\to_{R,\mu}$ with respect to a TRS R is defined inductively by

- if $\ell \to r \in R$ and ρ is a substitution, then $\ell\rho \to_{R,\mu} r\rho$;
- if $i \in \mu(f)$ and $t_i \to_{R,\mu} t'_i$ and $t'_j = t_j$ for all $j \neq i$, then $f(t_1, \ldots, t_n) \to_{R,\mu} f(t'_1, \ldots, t'_n)$.

In our setting we choose μ by $\mu(c) = \{1, \ldots, \mathsf{ar}(d, c)\}$ for all $c \in \mathcal{C}$, and $\mu(f) = \{1, \ldots, \mathsf{ar}(f)\}$ for all $f \in \Sigma_d \cup \Sigma_s$, where we write $\mathsf{ar}(f) = \mathsf{ar}(d, f) + \mathsf{ar}(s, f)$ for $f \in \Sigma_s$. In the rest of this paper the only instance of context-sensitive rewriting we consider is with respect to this particular μ, which is left implicit from now on. So in μ-rewriting, rewriting inside s-arguments of constructor symbols is disallowed, and is allowed in all other positions. A TRS is called *μ-terminating* if μ-rewriting is terminating.

Theorem 4.1. *Let $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ be a proper specification for which $R_s \cup R_d$ is μ-terminating for μ as defined above. Then the specification is productive.*

Proof. We define a ground μ-normal form to be a ground term that can not be rewritten by μ-rewriting. We prove the following claim by induction on t:

Claim: If t is a ground μ-normal form of sort s, then the $\mathsf{root}(t) \in \mathcal{C}$.

Assume $\mathsf{root}(t) \notin \mathcal{C}$. Then $t = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for $f \in \Sigma_s$, $u_1, \ldots, u_n$ are of sort d, and $t_1, \ldots, t_m$ are of sort s. Since $\mu(f) = \{1, \ldots, n + m\}$, they are all ground μ-normal forms. So $u_1, \ldots, u_n \in \mathcal{D}$. By the induction hypothesis all t_i have their roots in $\mathcal{C}$. Since t_i is a μ-normal form and the arguments of sort d are in $\mu(c)$ for every $c \in \mathcal{C}$, the arguments of t_i of sort d are all in $\mathcal{D}$. Due to the shape of the rules now a rule is applicable on t on the root level, so satisfies the restriction of μ-rewriting, contradicting the assumption that t is a μ-normal form. This concludes the proof of the claim.

According to Proposition 3.3 for productivity we have to prove that every ground term t of sort s rewrites to a term having its root in $\mathcal{C}$. Apply μ-rewriting on t as long as possible. Due to μ-termination this will end in a term on which μ-rewriting is not possible, so a ground μ-normal form. Due to the claim this ground μ-normal form has its root in $\mathcal{C}$. ∎

Example 4.2. Consider the following stream specification

$$\mathsf{ones} \to 1 : \mathsf{ones} \qquad \begin{array}{rcl} \mathsf{f}(0 : \sigma) & \to & \mathsf{f}(\sigma) \\ \mathsf{f}(1 : \sigma) & \to & 1 : \mathsf{f}(\sigma) \end{array}$$

Productivity for all ground terms including $\mathsf{f}(\mathsf{ones})$ follows from Theorem 4.1: entering this rewrite system in the tool AProVE [8] or μ-Term [13] together with the context-sensitivity information that rewriting is disallowed in the second argument of ':' fully automatically yields a proof of context-sensitive termination. Alternatively, by entering this specification in our tool yields exactly the same proof.

In this specification f is the stream function that removes all zeros. So productivity depends on the fact that the stream of all zeros does not occur as the interpretation of a subterm of any ground term in this specification. For instance, by adding the rule $\mathsf{zeros} \to 0 : \mathsf{zeros}$ the specification is not productive any more as $\mathsf{f}(\mathsf{zeros})$ does not rewrite to a term having a constructor as its root.

This also shows the difference between our requirement of productivity of all finite ground terms and the requirement in [11, 12] of productivity of all terms, including infinite terms. There this example is not productive on the infinite term representing the stream of all zeros. Finally we mention that the technique from [4] fails to prove productivity for $\mathsf{f}(\mathsf{ones})$, since the specification is not data obliviously productive.

Example 4.3. We specify the sorted stream of Hamming numbers: all positive natural numbers that are not divisible by other prime numbers than 2, 3 and 5. Here $\mathcal{D} = \{\mathsf{s}^n(0) \mid n \geq 0\}$. For $+$ and $*$ we have the standard rules, we also need comparison cmp for which $\mathsf{cmp}(n, m)$ yields 0 if $n = m$, $\mathsf{s}(0)$ if $n > m$ and $\mathsf{s}(\mathsf{s}(0))$ if $n < m$. So R_d consists of the rules

$$\begin{array}{rcl} x + 0 & \to & x \\ x + \mathsf{s}(y) & \to & \mathsf{s}(x + y) \\ x * 0 & \to & 0 \\ x * \mathsf{s}(y) & \to & (x * y) + x \end{array} \qquad \begin{array}{rcl} \mathsf{cmp}(0, 0) & \to & 0 \\ \mathsf{cmp}(\mathsf{s}(x), 0) & \to & \mathsf{s}(0) \\ \mathsf{cmp}(0, \mathsf{s}(x)) & \to & \mathsf{s}(\mathsf{s}(0)) \\ \mathsf{cmp}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{cmp}(x, y) \end{array}$$

For R_s we need a function mul to multiply a stream element-wise by a number, a function mer for merging two sorted streams, and an auxiliary function f. Finally we have a constant h for the sorted stream of Hamming numbers. The rules of R_s read:

$$\begin{array}{rcl} \mathsf{mul}(x, y : \sigma) & \to & x * y : \mathsf{mul}(x, \sigma) \\ \mathsf{mer}(x : \sigma, y : \tau) & \to & \mathsf{f}(\mathsf{cmp}(x, y), x : \sigma, y : \tau) \\ & & \end{array} \qquad \begin{array}{rcl} \mathsf{f}(0, x : \sigma, y : \tau) & \to & x : \mathsf{mer}(\sigma, \tau) \\ \mathsf{f}(\mathsf{s}(0), \sigma, y : \tau) & \to & y : \mathsf{mer}(\sigma, \tau) \\ \mathsf{f}(\mathsf{s}(\mathsf{s}(x)), y : \sigma, \tau) & \to & y : \mathsf{mer}(\sigma, \tau) \end{array}$$

$$\mathsf{h} \to \mathsf{s}(0) : \mathsf{mer}(\mathsf{mer}(\mathsf{mul}(\mathsf{s}^2(0), \mathsf{h}), \mathsf{mul}(\mathsf{s}^3(0), \mathsf{h})), \mathsf{mul}(\mathsf{s}^5(0), \mathsf{h}))$$

Now we have a proper stream specification, being the folklore functional program for generating Hamming numbers, up to notational details. Productivity is proved fully automatically by our tool: μ-Term [13] is called together with the context-sensitivity information that rewriting is disallowed in the second argument of ':', yielding a proof of context-sensitive termination. So by Theorem 4.1 productivity can be concluded.

For completeness we mention that the tool of [6, 4] also finds a proof of productivity of h in this example.

Example 4.4. The Calkin–Wilf tree [3] is a binary tree in which every node is labeled by a pair of natural numbers. The root is labeled by $(1, 1)$, and every node labeled by (m, n) has children labeled by $(m, m + n)$ and $(m + n, n)$. It can be proved that for all natural numbers $m, n > 0$ that are relatively prime the pair (m, n) occurs exactly once as a label of a node, and no other pairs occur. So the labels of the nodes represent positive rational numbers, and every positive rational number m/n occurs exactly once as a pair (m, n). There is one constructor b with $\mathsf{ar}(d, \mathsf{b}) = \mathsf{ar}(s, \mathsf{b}) = 2$. From Example 4.3 we take the data set $\mathcal{D}$ consisting of the natural numbers, and also the symbol $+$ and its two rules. Now the Calkin–Wilf tree c is defined by

$$\mathsf{c} \to \mathsf{f}(\mathsf{s}(0), \mathsf{s}(0)), \quad \mathsf{f}(x, y) \to \mathsf{b}(x, y, \mathsf{f}(x, x + y), \mathsf{f}(x + y, y)).$$

Our tool proves productivity of this specification by calling μ-Term [13] that proves context-sensitive termination, hence proving productivity by Theorem 4.1.

Theorem 4.1 can be seen as a strengthening of Theorem 3.4: if all roots of right-hand sides of rules from R_s are in $\mathcal{C}$ then $R_s \cup R_d$ is μ-terminating, as is shown in the following proposition.

Proposition 4.5. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ be a proper specification in which for every $\ell \to r$ in R_s the term r is not a variable and $\mathsf{root}(r) \in \mathcal{C}$. Then $R_s \cup R_d$ is μ-terminating.*

Proof. Assume there exists an infinite μ-reduction. For every term in this reduction count the number of symbols from Σ_s that are on allowed positions. Due to the assumptions by every R_d-step this number remains the same, while by every R_s-step this number decreases by one. So this reduction contains only finitely many R_s-steps. After these finitely many R_s-steps an infinite R_d-reduction remains, contradicting the assumption that R_d is terminating. ■

The reverse direction of Theorem 4.1 does not hold, as is illustrated in the next example.

Example 4.6. Consider the proper (stream) specification $(\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$, where $\Sigma_d = \{0, 1\}$, $R_d = \emptyset$, $\mathcal{C} = \{:\}$ with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$, and R_s being the below TRS:

$$\begin{array}{rcl} \mathsf{p} & \to & \mathsf{zip}(\mathsf{alt}, \mathsf{p}) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

This specification is productive, as we will see later in Example 5.2. However, it admits an infinite context-sensitive reduction $\mathsf{p} \to \mathsf{zip}(\mathsf{alt}, \mathsf{p})$ which is continued by repeatedly reducing the redex p.

The stream p describes the sequence of right and left turns in the well-known *dragon curve*, obtained by repeatingly folding a paper ribbon in the same direction.

5. Transformations for Proving Productivity

To be able to handle examples like the above, we investigate transformations of such specifications for which productivity of the original system can be concluded from productivity of the transfomed one. Whenever productivity of a specification cannot be determined directly, then we apply one of these transformations and try to prove productivity of the transformed specification, instead.

One such transformation is the reduction of right-hand sides, that is, a rule $\ell \to r$ of R_s is replaced by $\ell \to r'$ for a term r' satisfying $r \to^*_{R_s \cup R_d} r'$. Write $R = R_s \cup R_d$, and write R' for the result of this replacement. Then by construction we have $\to_{R'} \subseteq \to^+_R$, and $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$, that is, every $\to_R$-step can be followed by zero or more $\to_R$-steps to obtain a $\to_{R'}$-step. We present our theorems in this more general setting such that they are applicable more generally than only for reduction of right-hand sides.

Theorem 5.1. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ and $\mathcal{S}' = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R'_s)$ be proper specifications satisfying $\to_{R'} \subseteq \to^+_R$ for $R = R_s \cup R_d$ and $R' = R'_s \cup R_d$. If $\mathcal{S}'$ is productive, then $\mathcal{S}$ is productive, too.*

Proof. Let $\mathcal{S}'$ be productive, i.e., every ground term t of sort s admits a reduction $t \to^*_{R'} t'$ for which $\mathsf{root}(t') \in \mathcal{C}$. Then by $\to_{R'} \subseteq \to^+_R$ we conclude $t \to^*_R t'$, proving productivity of $\mathcal{S}$. ■

Example 5.2. We apply this theorem to Example 4.6. Observe that we can rewrite the right-hand side of the rule $\mathsf{p} \to \mathsf{zip}(\mathsf{alt}, \mathsf{p})$ as follows:

$$\mathsf{zip}(\mathsf{alt}, \mathsf{p}) \to \mathsf{zip}(0 : 1 : \mathsf{alt}, \mathsf{p}) \to 0 : \mathsf{zip}(\mathsf{p}, 1 : \mathsf{alt})$$

So we may transform our specification by replacing R_s by the TRS R'_s consisting of the following rules:

$$\begin{array}{rcl} \mathsf{p} & \to & 0 : \mathsf{zip}(\mathsf{p}, 1 : \mathsf{alt}) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Clearly, this is a proper specification that is productive due to Theorem 3.4. Now productivity of the original specification follows from Theorem 5.1 and $\to_{R'_s} \subseteq \to^+_{R_s}$. Our tool finds exactly this proof.

Concluding productivity of the original system from productivity of the transformed system is called *soundness*, the converse is called *completeness*. The following example shows the incompleteness of Theorem 5.1.

Example 5.3. Consider the two proper (stream) specifications $\mathcal{S}$ and $\mathcal{S}'$ defined by

$$R_s: \quad \begin{array}{rcl} \mathsf{c} & \to & \mathsf{f}(\mathsf{c}) \\ \mathsf{f}(\sigma) & \to & 0 : \sigma \end{array} \qquad\qquad R'_s: \quad \begin{array}{rcl} \mathsf{c} & \to & \mathsf{f}(\mathsf{c}) \\ \mathsf{f}(x : \sigma) & \to & 0 : x : \sigma \end{array}$$

Here $\mathcal{C} = \{:\}$, $R_d = \emptyset$, $\Sigma_d = \{0\}$. Since $\mathsf{c} \to_R \mathsf{f}(\mathsf{c}) \to_R 0 : \mathsf{c}$ and $\mathsf{f}(\cdots) \to_R 0 : \cdots$ we conclude productivity of $\mathcal{S}$, as c and f are the only symbols in Σ_s.

For the TRS R'_s we have that $\to_{R'_s} \subseteq \to^+_{R_s}$, since any step with the rule $\mathsf{f}(x : \sigma) \to 0 : x : \sigma$ of R'_s can also be done with the rule $\mathsf{f}(\sigma) \to 0 : \sigma$ of R_s. However, $\mathcal{S}'$ is not productive, as the only reduction starting in c is $\mathsf{c} \to \mathsf{f}(\mathsf{c}) \to \mathsf{f}(\mathsf{f}(\mathsf{c})) \to \cdots$ in which the root is never in $\mathcal{C}$.

Next we prove that with the extra requirement $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$, as holds for reduction of right-hand sides, we have both soundness and completeness.

Theorem 5.4. *Let $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R_s)$ and $\mathcal{S}' = (\Sigma_d, \Sigma_s, \mathcal{C}, R_d, R'_s)$ be proper specifications satisfying $\to_{R'} \subseteq \to^+_R$ and $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$ for $R = R_s \cup R_d$ and $R' = R'_s \cup R_d$.*

Then $\mathcal{S}$ is productive if and only if $\mathcal{S}'$ is productive.

Proof. The "if" direction follows from Theorem 5.1.

For the "only-if" direction first we prove the following claim:

> **Claim:** If $t \to_R t'$ and $t \to^*_R t''$, then there exists a term v satisfying $t' \to^*_R v$ and $t'' \to^*_{R'} v$.

Let $t \to_R t'$ be an application of the rule $\ell \to r$ in R, so $t = C[\ell\rho]$ and $t' = C[r\rho]$ for some C, ρ. According to the Parallel Moves Lemma ([17], Lemma 4.3.3, page 101), we can write $t'' = C''[\ell\rho_1, \ldots, \ell\rho_n]$, and t', t'' have a common R-reduct $C''[r\rho_1, \ldots, r\rho_n]$. Due to $\ell\rho_i \to_R r\rho_i$ and $\to_R \subseteq \to_{R'} \cdot \leftarrow^*_R$ there exist t_i satisfying $\ell\rho_i \to_{R'} t_i$ and $r\rho_i \to^*_R t_i$, for all $i = 1, \ldots, n$. Now choosing $v = C''[t_1, \ldots, t_n]$ proves the claim.

Using this claim, by induction on the number of $\to_R$-steps from t to t' one proves the generalized claim: If $t \to^*_R t'$ and $t \to^*_R t''$, then there exists a term v satisfying $t' \to^*_R v$ and $t'' \to^*_{R'} v$.

Let t be an arbitrary ground term of sort s. Due to productivity of $\mathcal{S}$ there exists t' satisfying $t \to^*_R t'$ and $\mathsf{root}(t') \in \mathcal{C}$. Applying the generalized claim for $t'' = t$ yields a term

v satisfying $t' \to_R^* v$ and $t \to_{R'}^* v$. Since $\mathsf{root}(t') \in \mathcal{C}$ and $t' \to_R^* v$ we obtain $\mathsf{root}(v) \in \mathcal{C}$. Now $t \to_{R'}^* v$ implies productivity of $\mathcal{S}'$. ∎

Example 5.3 generalizes to a general application of Theorem 5.1 other than rewriting right-hand sides as follows. Assume a rule from R_s in a proper transformation contains an s-variable σ in the left-hand side being an argument of the root. Then for every $c \in \mathcal{C}$ this rule may be replaced by an instance of the same rule, obtained by replacing σ by $c(x_1, \ldots, x_n, \sigma_1, \ldots, \sigma_m)$, where $\mathsf{ar}(d, c) = n, \mathsf{ar}(s, c) = m$. If this is done simultaneously for every $c \in \mathcal{C}$, so replacing the original rule by $\#\mathcal{C}$ instances, then the result is again a proper specification. Also the requirements of Theorem 5.1 hold, even $\to_{R'} \subseteq \to_R$. We show this transformation by an example.

Example 5.5. We want to analyze productivity of the following variant of Example 4.6, in which p has been replaced by a stream function, and R_s is the below TRS:

$$\begin{array}{rcl} \mathsf{p}(\sigma) & \to & \mathsf{zip}(\sigma, \mathsf{p}(\sigma)) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Proving productivity by Theorem 3.4 fails. Also proving productivity with the technique of Theorem 4.1 fails, since there exists the infinite context-sensitive reduction

$$\mathsf{p}(\mathsf{alt}) \to \mathsf{zip}(\mathsf{alt}, \underbrace{\mathsf{p}(\mathsf{alt})}) \to \ldots.$$

Furthermore, reducing the right-hand side of $\mathsf{p}(\sigma) \to \mathsf{zip}(\sigma, \mathsf{p}(\sigma))$ can only be done by applying the first rule, not creating a constructor as the root of the right-hand side. What blocks rewriting using the zip rule is the variable σ in the first argument of zip. Therefore, we apply Theorem 5.1 as sketched above, note that $\mathcal{C} = \{:\}$, and replace the rule $\mathsf{p}(\sigma) \to \mathsf{zip}(\sigma, \mathsf{p}(\sigma))$ by the single rule $\mathsf{p}(x : \sigma) \to \mathsf{zip}(x : \sigma, \mathsf{p}(x : \sigma))$ to obtain the TRS R_s'. This now allows us to rewrite the new right-hand side by the zip rule, replacing the previous rule by $\mathsf{p}(x : \sigma) \to x : \mathsf{zip}(\mathsf{p}(x : \sigma), \sigma)$, i.e., we obtain the TRS R_s'' consisting of the following rules:

$$\begin{array}{rcl} \mathsf{p}(x : \sigma) & \to & x : \mathsf{zip}(\mathsf{p}(x : \sigma), \sigma) \\ \mathsf{alt} & \to & 0 : 1 : \mathsf{alt} \\ \mathsf{zip}(x : \sigma, \tau) & \to & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Productivity of R_s'' follows from Theorem 3.4. This implies productivity of R_s' due to Theorem 5.1 which in turn implies productivity of our initial specification $\mathcal{S}$, again due to Theorem 5.1. Our tool finds exactly the proof as given here.

Example 5.6. For stream computations it is often natural also to use finite lists. The data structure combining streams and finite lists is obtained by choosing $\mathcal{C} = \{:, \mathsf{nil}\}$, with $\mathsf{ar}(d, :) = \mathsf{ar}(s, :) = 1$ and $\mathsf{ar}(d, \mathsf{nil}) = \mathsf{ar}(s, \mathsf{nil}) = 0$, as mentioned in Example 2.3. An example using this is defining the sorted stream $p = 1 : 2 : 2 : 3 : 3 : 3 : 4 : \cdots$ of natural numbers, in which n occurs exactly n times for every $n \in \mathbf{N}$. This stream can be defined by a specification not involving finite lists, but here we show how to do it in this extended data structure based on standard operations like conc. Apart from conc we use copy, for which $\mathsf{copy}(k, n)$ is the finite list of k copies of n, for $k, n \in \mathbf{N}$, and a function f for generating $\mathsf{p} = \mathsf{f}(0)$. Taking $\mathcal{D}$ to be the set of ground terms over $\{0, \mathsf{s}\}$ and $R_d = \emptyset$, we choose R_s to

consist of the following rules:

$$\begin{array}{rclrcl} \mathsf{p} & \to & \mathsf{f}(0) & \mathsf{f}(x) & \to & \mathsf{conc}(\mathsf{copy}(x,x),\mathsf{f}(\mathsf{s}(x))) \\ \mathsf{copy}(\mathsf{s}(x),y) & \to & y:\mathsf{copy}(x,y) & \mathsf{conc}(\mathsf{nil},\sigma) & \to & \sigma \\ \mathsf{copy}(0,x) & \to & \mathsf{nil} & \mathsf{conc}(x:\sigma,\tau) & \to & x:\mathsf{conc}(\sigma,\tau) \end{array}$$

Note that productivity of this system is not trivial: if the rule for f is replaced by $\mathsf{f}(x) \to \mathsf{conc}(\mathsf{copy}(x,x),\mathsf{f}(x))$, then the system is not productive.

Productivity cannot be proved directly by Theorem 3.4 or Theorem 4.1; context-sensitive termination does not even hold for the single f rule. However by replacing the f rule by the two instances

$$\mathsf{f}(0) \to \mathsf{conc}(\mathsf{copy}(0,0),\mathsf{f}(\mathsf{s}(0))) \quad \text{and} \quad \mathsf{f}(\mathsf{s}(x)) \to \mathsf{conc}(\mathsf{copy}(\mathsf{s}(x),\mathsf{s}(x)),\mathsf{f}(\mathsf{s}(\mathsf{s}(x)))),$$

and then applying rewriting right-hand sides by which these two rules are replaced by

$$\mathsf{f}(0) \to \mathsf{f}(\mathsf{s}(0)) \quad \text{and} \quad \mathsf{f}(\mathsf{s}(x)) \to \mathsf{s}(x):\mathsf{conc}(\mathsf{copy}(x,\mathsf{s}(x)),\mathsf{f}(\mathsf{s}(\mathsf{s}(x))))$$

yields a proper specification for which context-sensitive termination is proved by AProVE [8] or μ-Term [13], proving productivity of the original example by Theorem 5.1 and Theorem 4.1. Our tool finds a similar proof as given here: right-hand sides were slightly more rewritten.

Example 5.7. We conclude this section by an example in binary trees, in which the nodes are labeled by natural numbers, so there is one constructor $\mathsf{b} : d \times s^2 \to s$ and $\mathcal{D}$ consists of ground terms over $\{0, \mathsf{s}\}$. The rules are

$$\begin{array}{rclrcl} \mathsf{c} & \to & \mathsf{b}(0,\mathsf{f}(\mathsf{g}(0),\mathsf{left}(\mathsf{c})),\mathsf{g}(0)) & \mathsf{left}(\mathsf{b}(x,xs,ys)) & \to & xs \\ \mathsf{g}(x) & \to & \mathsf{b}(x,\mathsf{g}(\mathsf{s}(x)),\mathsf{g}(\mathsf{s}(x))) & \mathsf{f}(\mathsf{b}(x,xs,ys),zs) & \to & \mathsf{b}(x,ys,\mathsf{f}(zs,xs)) \end{array}$$

To get an impression of the hardness of this example, observe that f and left are similar to zip and tail for streams, respectively, and the recursion in the rule for c has the flavor of $\mathsf{c} \to 0 : \mathsf{zip}(\cdots,\mathsf{tail}(\mathsf{c}))$. Our tool proves productivity by Theorem 5.1 and Theorem 4.1, by first rewriting right-hand sides and then proving context-sensitive termination.

6. Implementation

We have implemented a tool to check productivity of proper specifications using the techniques presented in this paper. It is accessible via the web-interface

`http://pclin150.win.tue.nl:8080/productivity`.

The input format requires the following ingredients:

- the variables,
- the operation symbols with their types,
- the rewrite rules.

Details of the format can be seen from the examples that are available. All other information, like which symbols are in $\mathcal{C}$ is extracted by the tool from these ingredients.

As a first step, the tool checks that the input is indeed a proper specification. Checking syntactic requirements, such as no function symbol returning sort d has an argument of sort s, the TRS is 2-sorted and orthogonal, and the left-hand sides have the required shape, are all straightforward. However, to verify the last requirement of a proper specification, namely that the TRS is exhaustive, is a hard job if we allow $\mathcal{D}$ to be the set of ground normal forms

of any terminating orthogonal R_d. Instead we restrict to the class of proper specifications in which $\mathcal{D}$ consists of the constructor ground terms of sort d, i.e., the terms in $\mathcal{D}$ do not contain symbols occurring as root symbol in a left-hand side of a rule in R_d. To check whether this is the case, we use anti-matching as described in [14]. It can easily be shown that the normal forms of ground terms w.r.t. R_d are only constructor terms if and only if there is no anti-matching term that has a defined symbol as root and only terms built from constructors and variables as arguments. The idea of the proof is that such a term could be instantiated to a ground term, which is a normal form due to the anti-matching property. Then, checking exhaustiveness of R_s has to only consider constructor terms for both data and structure arguments.

To analyze productivity of a given proper specification, the tool first investigates whether Theorem 3.4 can be applied directly: it checks whether the roots of all right-hand sides are constructors. If this simple criterion does not hold, then it tries to show context-sensitive termination using the existing termination prover μ-Term, by which productivity will follow by Theorem 4.1.

If both of these first attempts fail then the tool tries to transform the given specification. Since rewriting of right-hand sides is both sound and complete, as was shown in Section 5, a productive specification can never be transformed into an unproductive one by this technique. Therefore, this is the first transformation to try. However, large right-hand sides often make it harder for termination tools to prove context-sensitive termination. Therefore, the tool tries to only rewrite positions on right-hand sides that appear to be needed to obtain a constructor prefix tree of a certain, adjustable depth. This is done by traversing the term in an outermost fashion and only trying to rewrite arguments if the possibly matching rules require a constructor for that particular argument. If at least one right-hand side could be rewritten, a new specification with the rewritten right-hand sides is created. Since rewriting of right-hand sides is not guaranteed to terminate, we limit the maximal number of rewriting steps. After rewriting the right-hand sides in this way, the tool again tries to prove productivity of the transformed TRS using our basic techniques.

As shown in Examples 5.5 and 5.6, it can be helpful to replace a variable by all constructors of its sort applied to variables. Therefore, in case productivity could not be shown so far, it is tried to instantiate a variable on a position of a right-hand side that is required by the rules for the defined symbol directly above it. Then the instantiated right-hand sides are rewritten again to obtain new specifications for which productivity is analyzed further.

The described transformations are applied in the order of their presentation a number of times. If a set limit of applications of transformations is reached, the tool finally tries to rewrite to deeper context-prefixes on right-hand sides and does a final check for productivity, using a larger timeout value.

Using these heuristics the tool is able to automatically prove productivity of all productive examples presented in this paper. This especially includes the example of a stream specification given in the following section, which could not be proved to be productive by any other automated technique we are aware of.

7. Conclusions and Related Work

We have presented new techniques to prove productivity of specifications of infinite objects like streams. Until now several techniques were developed for proving productivity of

stream specifications, but not for other infinite data structures like infinite trees or the combination of streams and finite lists. In this paper we gave several examples of applying our techniques to these infinite data structures. We implemented a tool by which productivity of all of these examples could be proved fully automatically. For the non-stream examples there are hardly other techniques to compare. For streams there are examples where our technique outperforms all earlier techniques. For instance, the techniques from [6, 4] fail to prove productivity of Example 4.2. For this example the technique from [19] succeeds, but this technique fails as soon binary stream operations come in like zip. To our knowledge our technique is the first that can deal with productivity for $\mathsf{f}(\mathsf{p})$ of the specification consisting of the combination of Example 4.6 (describing the paper folding stream) and the two rules $\mathsf{f}(0 : \sigma) \to \mathsf{f}(\sigma)$, $\mathsf{f}(1 : \sigma) \to 1 : \mathsf{f}(\sigma)$. Our tool first performs rewriting of the right-hand side of the p-rule and then proves context-sensitive termination by μ-Term. Note the subtlety in this example: as soon as a ground term t can be composed of which the interpretation as a stream contains only finitely many ones, then the system will not be productive for $\mathsf{f}(t)$. So as a consequence we conclude that the paper folding stream p contains infinitely many ones, as the specification is productive for $\mathsf{f}(\mathsf{p})$.

Some ideas in this paper are related to earlier observations. In [10] the observation was made that if right-hand sides of stream definitions have ':' as its root, then well-definedness can be concluded, comparable to what we did by Theorem 3.4, and can be concluded from friendly-nestingness in [4]. A similar observation can be made about process algebra, where a recursive specification is called *guarded* if right-hand sides can be rewritten to a choice among terms all having a constructor on top, see e.g. [2], Section 5.5. In that setting every specification has at least one solution, while guardedness also implies there is at most one solution ([2], Theorem 5.5.11). So guardedness implies well-definedness, being of the flavor of combining Theorem 3.4 with rewriting right-hand sides. From both of these observations we obtain well-definedness, which is a slightly weaker notion than productivity. An investigation of well-definedness for stream specifications based on termination was made in [18]. We want to stress that productivity is strictly stronger than well-definedness, which is shown by the stream specification $\mathsf{c} \to \mathsf{f}(\mathsf{c})$, $\mathsf{f}(x : \sigma) \to 0 : \mathsf{c}$, being well-defined but not productive.

As far as we know the relationship of productivity with context-sensitive termination as expressed in Theorem 4.1 is new. Some ingredients of this relationship were given before in [19] where productivity of stream specifications was related to outermost termination and in [7] where outermost termination was related to context-sensitive termination. An alternative way to proceed would have been a further elaboration of combining the approaches from [19] and [7] to prove productivity: if the combination of the specification and a particular overflow rule is outermost terminating, then the specificiation is productive. Here for proving outermost termination the approach from [7] can be used. However, for examples like Example 4.3 this approach fails, even in combination with rewriting right-hand sides, while context-sensitive termination can be proved by standard tools, proving productivity by Theorem 4.1. For the other way around we only found examples where the direct approach is successful, too, in combination with rewriting right-hand sides. Apart from these experiments some intuition why our approach is to be preferred is the following. By the technique from [7] to prove outermost termination by proving context-sensitive termination of a transformed system, the size of the system increases dramatically. If the goal is to prove productivity, compared to the approach of this paper it is quite a detour to first transform the problem to outermost termination and then use such a strongly expanding transformation to relate it to context-sensitive termination, while it can be done directly without

such an expansion by Theorem 4.1. Summarizing, we do not expect that the power of our approach can be improved by extending it by trying to prove outermost termination of the specification extended by the overflow rule, neither for streams nor for other infinite data structures.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, UK, 1998.

[2] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 2009.

[3] N. Calkin and H. Wilf. Recounting the rationals. *American Mathematical Monthly*, 107(4):360–363, 2000.

[4] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-oblivious stream productivity. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, volume 5330 of *Lecture Notes in Computer Science*, pages 79–96. Springer-Verlag, 2008. Web interface tool: `http://infinity.few.vu.nl/productivity/`.

[5] J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of Fractran and productivity. In *Proceedings of the 22th Conference on Automated Deduction (CADE'09)*, volume 5663 of *Lecture Notes in Computer Science*, pages 371–387. Springer-Verlag, 2009.

[6] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J.W. Klop. Productivity of stream definitions. In *Proceedings of the Conference on Fundamentals of Computation Theory (FCT '07)*, volume 4639 of *Lecture Notes in Computer Science*, pages 274–287. Springer-Verlag, 2007.

[7] J. Endrullis and D. Hendriks. From outermost to context-sensitive rewriting. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 2009.

[8] J. Giesl et al. AProVE. Web interface and download: `http://aprove.informatik.rwth-aachen.de`.

[9] J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 14:329–427, 2004.

[10] R. Hinze. Functional pearl: streams and unique fixed points. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 189–200. ACM, 2008.

[11] A. Isihara. Productivity of algorithmic systems. In *SCSS 2008*, volume 08-08 of *RISC-Linz Report*, pages 81–95, 2008.

[12] A. Isihara. *Algorithmic Term Rewriting Systems.* PhD thesis, Free University Amsterdam, 2010.

[13] S. Lucas et al. μ-Term. Web interface and download: `http://zenon.dsic.upv.es/muterm/`.

[14] M. Raffelsieper and H. Zantema. A transformational approach to prove outermost termination automatically. In *Proceedings of the 8th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'08)*, volume 237 of *Electronic Notes in Theoretical Computer Science*, pages 3–21. Elsevier Science Publishers B. V. (North-Holland), 2009.

[15] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.

[16] J. G. Simonsen. The Π_2^0-completeness of most of the properties of rewriting systems you care about (and productivity). In R. Treinen, editor, *Proceedings of the 20th Conference on Rewriting Techniques and Applications (RTA)*, Lecture Notes in Computer Science. Springer, 2009.

[17] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 2003.

[18] H. Zantema. Well-definedness of streams by termination. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA'09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 164–178. Springer-Verlag, 2009.

[19] H. Zantema and M. Raffelsieper. Stream productivity by outermost termination. In *Proceedings of the 9th International Workshop in Reduction Strategies in Rewriting and Programming (WRS'09)*, volume 15 of *Electronic Proceedings in Theoretical Computer Science*, 2010.

This work is licensed under the Creative Commons Attribution Non-Commercial No Derivatives License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-nd/3.0/`.